Mastering CodeWarrior for Windows 95/NT

The Official Guide

CHAPTER

SEVEN

7

Compiling and Linking Your Code

- Choosing a Compiler
- Controlling a Compile
- Controlling a Link
- Cool Features
- Hands-On Exercises

Once you achieve a critical mass of code, it's time to get that code running so you can test it. To do so, you must compile and link the code into final form—be it executable, library, or dynamically linked library.

In CodeWarrior, this means you want to use the front-end (language) compiler to translate your code into the Intermediate Representation, and a back-end (target) compiler to generate the corresponding processor instructions from the IR. Then you link the resulting pieces together into final form. Of course, you never see the IR; it does its work invisibly. Nevertheless, it is there, the crossroads where languages and targets meet.

This chapter covers both kinds of compilers—front-end and back-end—as well as the linkers, and shows you how to use them. Along the way, you will also encounter a variety of related topics, such as viewing disassembly, preprocessing C/C++ code, and more.

Choosing a Compiler

The first step in the entire compiling process is deciding which compiler to use, for both target and language. That decision occurs long before you issue a command to compile code. Let's look at the plugin compiler architecture again, and see how you pick your compilers.

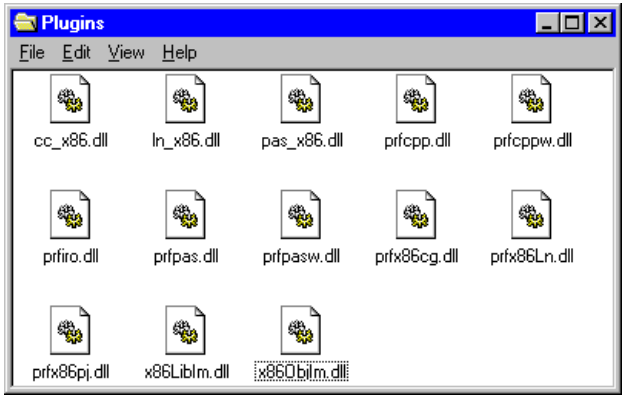
Some of what this section covers you have already encountered. You explored the inner workings of the compiler architecture and the Intermediate Representation in Chapter 1. We discussed languages and targets briefly in Chapter 3 on the project manager. Although it may seem like we are back at the beginning of our journey, the process is more an upward spiral than a circle.

You know a lot more about CodeWarrior now than you did then. Fundamentally, CodeWarrior is a development environment designed to make it as painless as possible to get your code processed through a compiler and linker so you can turn it into a final product. Compilers and linkers are what CodeWarrior is all about. So, having met them up front, it's time to meet them again with a new insight and deeper understanding of how they work, and their significance in the development process.

Plugin Tools

You are familiar with the CodeWarrior plugin architecture for compilers and linkers. All the CodeWarrior plugin compilers can be found in the Plugins folder, located inside the CodeWarrior folder. Figure 7.1 shows this directory as it appears in the Windows desktop.

FIGURE 7.1
The Plugins folder contains all the compilers, linkers, and settings panels.



You can wander through this directory if you'd like, but it is nothing more than a storage bin so CodeWarrior knows where to find the tools. You do not work with the compilers and linkers by launching them directly from the desktop. The only time you need to use this folder at all is when you want to add a new compiler or linker to

CodeWarrior. Then you place it, and the settings panels associated with that tool, in the Plugins folder.

Any panel in this folder shows up in the appropriate preferences or settings dialog box, and gives you access to the settings for each compiler and linker. You learned about the preferences and settings panels back in Chapter 3 on the project manager. We're going to examine some of these panels again, as a prelude to exploring the compiling process in CodeWarrior.

TIP

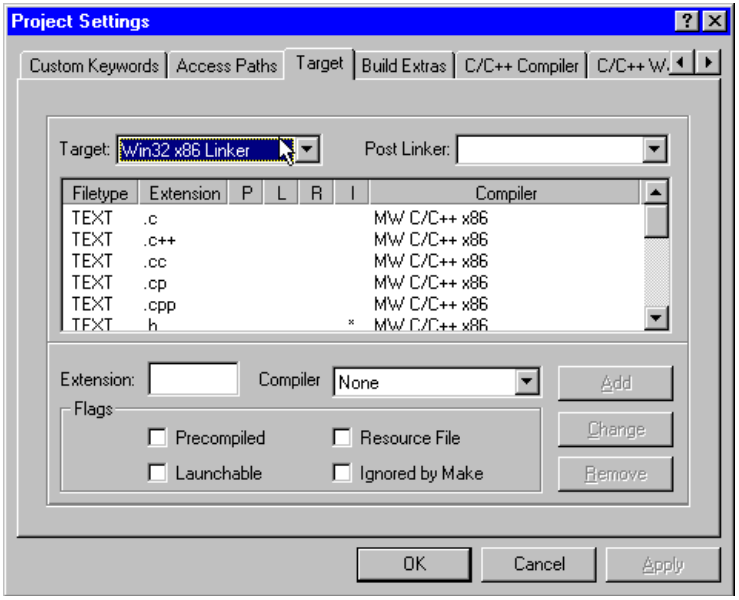
Metrowerks has made its plugin compiler and linker interface public. The CodeWarrior *Plugin API Guide* describes the plugin Application Programming Interface (API). These are the function calls CodeWarrior understands and uses to communicate with external plugin tools. The *Plugin API Guide* is useful for compiler engineers who want to create CodeWarrior-compatible plugin compilers, linkers, and panels. At the time of this writing, the manual is aimed at Mac OS-hosted tools. While the final implementation of the Windows-hosted plugin API will certainly be different in details, the *Plugin API Guide* is still extremely useful as a reference to the available functionality and will soon be available for Windows-hosted tools.

Now that you've seen where the pieces are, let's see how to choose both a target and a language compiler.

Choosing a Target Compiler

One half of the CodeWarrior compiler architecture is the target. You must explicitly choose a target compiler for a particular processor and operating system. To choose a target, you use the Target settings panel we discussed in Chapter 3. Figure 7.2 shows this panel. To pick a target compiler, you use the Target pop-up menu at the top of the panel. The list of available target compilers appears in this menu. Just select the target you want. Picking a target compiler is that easy.

FIGURE 7.2
Use the Target settings panel to pick a target.



The precise name of the target may actually specify a linker, and in fact you are choosing a linker as well, as you'll see when we discuss how to choose a linker. The most critical tool you're selecting when you pick a target is the code-generating back-end compiler.

CodeWarrior filters your available settings based on the chosen target. For example, if you choose Java as a target (the Java Virtual Machine), then the C/C++ and Pascal language and warnings panels disappear (because CodeWarrior does not support programming in those languages for the JVM). Similarly, the list of file types and extensions you see in the Target settings panel also changes. We'll discuss this part of the panel in just a bit when we talk about picking a language compiler.

So the first thing you should do when you create a project is set the target. Of course, if you use project stationery, the target is already set properly. However, it is important to know what's going on, because you have to create your own stationery, or you may decide to switch targets at some point.

WARNING

Changing targets in midstream is usually not a good thing to do. Most projects require target-dependent libraries of some kind. Changing the target does not replace those libraries with a new set of libraries appropriate for the new target. If you change targets, remember to look through your project for any files that would be inappropriate for the new target.

Choosing a Language Compiler

Where the process of choosing a target is explicit, the process of choosing a language compiler is subtle. In fact, you choose a language compiler automatically when you name a source file. Nevertheless, it is important that you understand what's going on behind the scenes. While an automatic or default mechanism is fine for most cases, a time may come when, as master of the tools, you decide you want to control precisely which language you use for which file. For that to happen, you must understand how the mechanism works.

The process starts in the Target settings panel, shown in Figure 7.3. You just saw how to use this panel to set a target. Let's take a closer look at this panel to see how you use it to pick a language as well.

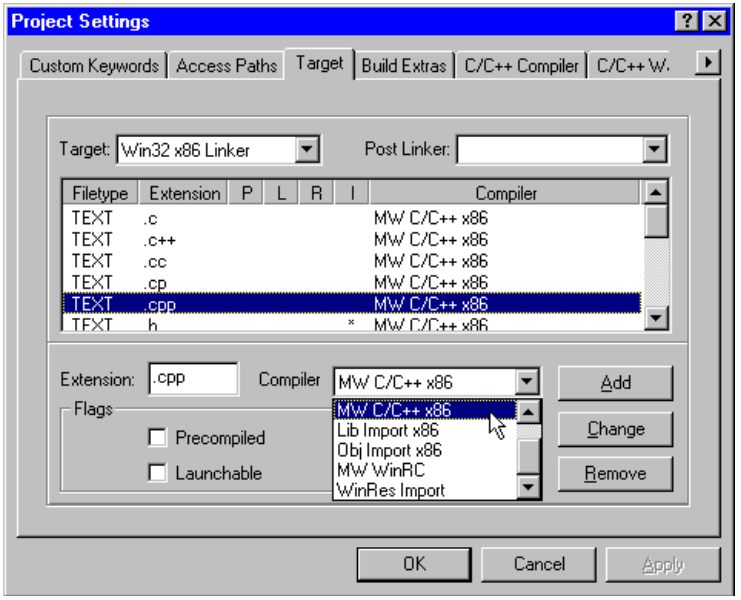
The area of interest is the scrolling list of files, and the buttons below them in this panel. The list of files represents every different kind of file CodeWarrior recognizes for this project. Because this is a settings panel, it applies only to the particular active project. If no project is open, changes you make to this list become the default settings for any new project created without using stationery. If you start a project using stationery, that project's list of file types applies to the new project.

The information in this list appears in seven columns. In order from left to right they are:

- File type
- File name extension
- Precompiled flag

FIGURE 7.3

The Target settings panel connects files with language compilers based on naming extensions.



- Launchable flag
- Resource File flag
- Ignored By Make flag
- Compiler

The list shows you which compiler interprets code for each kind of file using each naming extension. As you select an item in the list, the information in the controls below the list (the pop-up menu, check boxes, and text field in the panel) changes to reflect the information for the selected item.

You can change the compiler associated with a particular naming extension very simply. First, select the entry in the list you want to change. The Compiler pop-up menu lists all available compilers. Just pick a new compiler for that kind of file. Then click the Change button to change the item. You can add new file types and extensions, change existing information, or remove an item from the list altogether.

If you want to add a new kind of file to the list, specify the naming extension, the compiler, and the values for the four flags. Then click the Add button to add a new entry to the list. To remove an item from the list, select it and click the Remove button.

The flags tell CodeWarrior something about how you want the file handled. The Precompiled flag is useful for C/C++ precompiled header files. The .pch extension is a common naming convention for such a file. We'll discuss precompiled headers a little later in this chapter. Similarly, the Resource File flag tells CodeWarrior this particular kind of file is a resource. If the naming extension represents a resource file, set this flag.

NOTE

The Resource File flag is significant when you target Mac OS, which uses resources heavily. Saying a file is a resource file tells the Project Manager to merge the Mac OS resource data into the final build. You don't need to compile these resources—they are ready to go. However, when targeting Windows, you use a .rc file that contains resource commands, much like a script. The .rc file extension is mapped to the Metrowerks WinRC compiler for compilation, and the output .res file is linked into the project. You should not mark the .rc file extension as a resource file.

The Launchable flag tells CodeWarrior that this particular kind of file can be opened by some application (launched). If a file is launchable, when you double-click the file name in the project manager window, CodeWarrior attempts to launch the application that created the file, and then open the file. You'll notice as you scroll through the list of file types in this panel that all the source file types—.c, .cp, .c++, .p, and so forth—are not marked as launchable. CodeWarrior opens these files itself, so there is no need to launch a separate application.

Finally, the Ignored By Make flag tells CodeWarrior that the linker should ignore this file. Once again, a quick scroll through the list shows you very few if any files that fit into this category. One kind of file that could be ignored by the linker is the C/C++ header file using the extension .h.

When you finish specifying information in the File Type, Extension, Compiler, and Flags items, click the Add or Change buttons to create a new entry or modify an existing entry. For example, if you are working on a Java project, you may very well want to have an HTML file attached to your project—perhaps the page on which an applet is expected to run, or a simple bit of HTML code so you can see the applet in CodeWarrior.

You don't want the HTML file to be compiled, you just want it available in the project so you can access it readily. You can add the .htm extension to the list, and set it to have no compiler. CodeWarrior recognizes that the project accepts files with the .htm naming extension, and lets you add them to the project. If you identify the HTML file type as launchable, CodeWarrior attempts to open a viewing application—which might be your favorite browser—when you double-click the file name in the project window. However, an HTML file is just a text file, and the CodeWarrior editor can open and modify it.

WARNING

CodeWarrior only recognizes file types and extensions in this list. If a file name extension does not match an entry in this list, you cannot add it to a CodeWarrior project. Sometimes you can inadvertently add a trailing space when naming a file, and then you can't figure out why CodeWarrior isn't recognizing the file. If you have trouble adding a file, keep this in mind and check your file name.

What does this all mean for choosing a language compiler? Simply this: When you save and name a file, use the right naming extension. The very act of naming the file chooses the language compiler you want to use for that file. By default, the list you see in this panel that relates naming extensions to compilers covers all the standard naming conventions. Table 7.1 lists the most important.

If you use a different naming convention, make sure you add your particular extension to the list. If you do this with no project open, you've set the CodeWarrior default values. If you do it with a project

TABLE 7.1 The relationship between compilers and name extensions

Compiler	Standard Naming Extensions
C/C++	.c, .cp, .cpp, .c++, .cc, .h, .pch, .pch++
Pascal	.p, .pas
Java	.java, .class

open, you can save the project as stationery so you don't have to do it again for every project.

Controlling a Compile

A great chef cares about not only the ingredients that go into a gourmet feast, but also about *how* the food is prepared. As a programmer, you care not only about the underlying source code, you also care about how the compiler interprets that code and generates instructions for the target processor.

In this section we take a look at how to control both the language compiler and the code generated by the target compiler. What you need to know before you get down into these details is how to control which files are compiled, and how to start a compile.

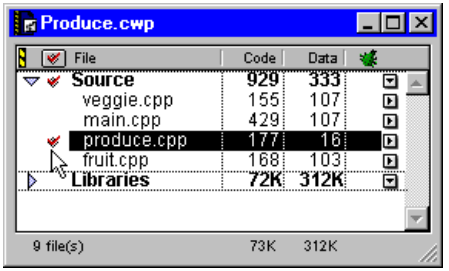
Deciding Which Files to Compile

The project window shows you which files are scheduled for compiling. Any file that the project manager thinks should be compiled has a check mark to the left of the name in the project window, as shown in Figure 7.4.

As we have discussed before, you can touch and untouch files at any time to control which files are compiled. Use the pop-up menu to the right of each file name, and choose the Touch or Untouch item from

FIGURE 7.4

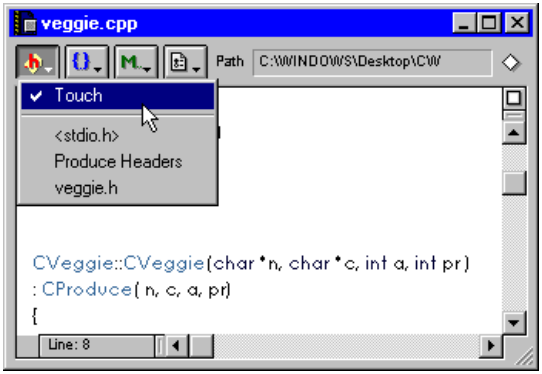
The Project Manager places a check mark next to each file to show it will be compiled.



that menu as appropriate. The Interface pop-up menu in an editor window has the same feature, as you can see in Figure 7.5.

FIGURE 7.5

The Interface pop-up menu lets you touch and untouch a file.



Of course, if you open a file and make a change, CodeWarrior automatically marks the file to be compiled. As a result, most of the time you do not have to manually touch or untouch files because CodeWarrior takes care of it for you. However, if you use a third-party editor (in which case CodeWarrior may not know a file has changed) or you want to recompile a file for some other reason, you can always mark the file yourself.

Issuing a Compile Command

CodeWarrior has several commands that cause one or more files to be compiled. You'll find each of these commands in the Project menu. They are:

- **Compile**—compiles the active window or selected file(s) in the project window
- **Bring Up To Date**—compiles all touched files in the project
- **Make**—brings up to date and links
- **Run**—makes and launches the executable
- **Debug**—makes and launches the executable under debugger control

As you can see, you can compile any individual file or selection of files using the Compile command. You can also compile all touched files with the Bring Up To Date command. In each case, the compiler generates the object code for that file, but does not link the final project together. As the compile progresses, you'll see the Code and Data numbers in the project window change.

NOTE

Object code is stored by the project manager. You can remove all object code from a project with the Remove Binaries command in the Project menu. All the Code and Data items in the project window will be set to zero. This of course means you must recompile the entire project again to build your final product.

The other three commands—Make, Run, and Debug—not only compile code, they also link the object code into a final product. We'll discuss those commands in just a little while when we discuss linking.

There are many reasons why you might want to compile an individual file or files without updating the entire project. For example, you might want to check the syntax of the file just to see if any errors exist before closing a file. A Check Syntax command is available, but the

Compile command does just as good a job, and if it finds no errors it gives you the object code as well. In the rest of this chapter you'll encounter additional reasons why compiling a single file can be a good thing.

Of course, any time you compile you may get errors or warnings. We'll talk about them a little later in the Cool Features section. You might not think of errors as cool things, but CodeWarrior's error handling mechanism certainly qualifies.

Now that you know how to pick which files are compiled and issue a compile command, let's look at how to control the compilers themselves.

Controlling Code Generation

When you pick a target, a whole series of additional panels relating to the compile process become important. For example, if you choose Win32x86 as your target (Windows programming), you have five panels of interest:

- **x86 Project**—choose project type, set up stack and heap size
- **WinRC Compiler**—specify a prefix file for the resource compiler
- **IR Optimizer**—choose optimizing strategy applied to the IR before code generation
- **x86 CodeGen**—control code generation and optimizing for the x86 chip
- **x86 Linker**—control the linker for Win32 code

If you pick another target, a different set of panels will interest you. The settings in each of these panels is really target-specific, so we will discuss the details of these panels later on in this book when we talk about writing code specifically for Windows 95/NT. Other chapters will cover the analogous panels for other targets—Mac OS and Java VM.

However, you should realize that these panels give you the low-level, down-in-the-gears control over how CodeWarrior generates your code. For example, you can specify the name of the final application and its heap size, how code is optimized, and much more.

While we are skipping the details for now so we can focus on compiling as it applies to all targets, keep in mind that when you work on a project, these are the kinds of panels you come to when you want to control the code generated by the compiler and linker.

Controlling a Language Compiler

Unlike the code generation issues related to a specific target, the front-end compilers are target-neutral. Earlier in this chapter you learned how to pick a language compiler for a particular file. In addition to choosing which language to use, you can also modify how the language compiler works.

You control the front-end compiler through the appropriate language settings panel. The language settings panels you have available depend upon the target you select, because CodeWarrior does not support programming in all languages for all targets. The panels you will typically have available to you may include panels for both the C/C++ and Pascal languages. (Java has no specific language settings panels.)

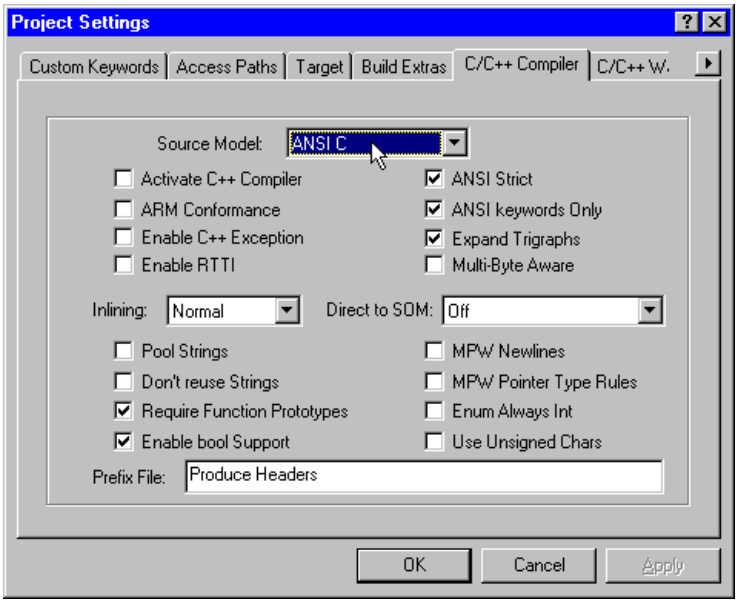
NOTE

Once again you can see the effects of the IR, another sign of the front-end/back-end architecture. Where the target settings control the back-end compiler, the language settings control the front-end compiler.

The settings you have available demonstrate the power of the Metrowerks compilers. Two panels are available for C/C++, and two for Pascal. Each language has a panel for the language compiler, and another for compiler warnings. Figure 7.6 shows the C/C++ Compiler panel. We'll discuss warnings later on in this chapter.

FIGURE 7.6

The C/C++ Compiler settings panel controls how this front-end compiler works.



A quick glance at the options in Figure 7.6 tells you that the CodeWarrior C/C++ compiler is very powerful. It supports C++ exceptions, run-time type identification, and compiling direct to the standard object model (SOM). It also provides a variety of options for implementing inline requests in your code.

The Source Model pop-up at the top of the panel contains several default options for this panel. The settings that conform to the ANSI C standard are shown. You can pick other code models, such as ANSI C++, or Apple C if you were doing C programming for the Mac OS. When you pick an option from the pop-up menu, the settings in the panel change to match what's required for that particular model. You can also set individual options as you wish.

Some of these options go right to the inner core of C++ programming, because the CodeWarrior compiler tracks the latest develop-

ments in the emerging C++ standard. A detailed explanation of the meaning or effect of each option in the C/C++ Language settings panel is really beyond the scope of this book, but don't be alarmed. The CodeWarrior documentation explains fully how the compiler works and what each of these settings does. If you have questions about any of these options, or you are unsure about what an option does, you'll find the answer in the *CodeWarrior C/C++/Asm Language Reference*.

TIP

The C/C++ compiler (when targeting Windows) also supports inline assembly language. When targeting Mac OS, the compiler does not currently support true inline assembly where you put in a few lines of assembly code in the middle of a regular C/C++ function. Instead, it supports *intrinsics* that you can use as a substitute for true inline assembly. The compiler also supports function-level assembly, where you create an entire function in assembly language for either 68K or PowerPC. Inline assembly sits high on the list of features soon to be added to the compilers. The details of assembly language support and intrinsics can be found in the *C/C++/Asm Language Reference* and in the *Targeting Win32* manuals.

The Pascal Compiler settings panel, as shown in Figure 7.7, demonstrates that the Pascal compiler is equally advanced. Once again, each option is fully detailed in the CodeWarrior documentation—in this case in the *CodeWarrior Pascal Language Reference*.

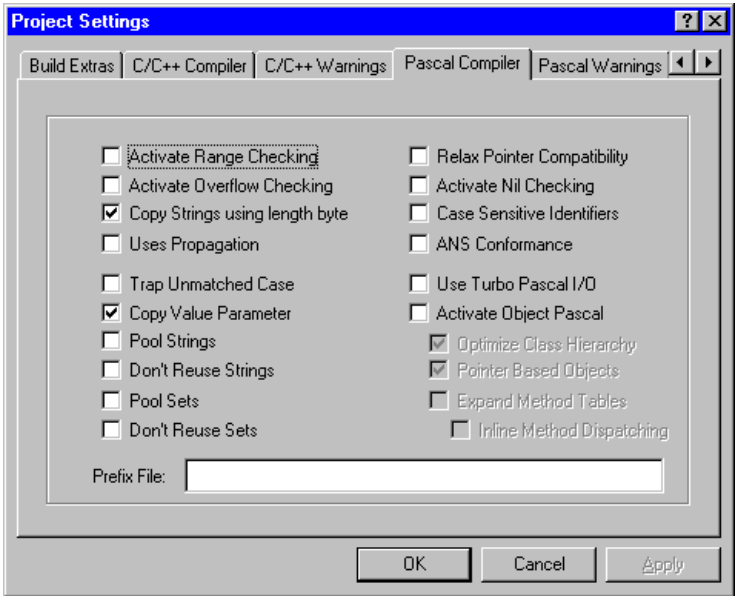
TIP

The Pascal compiler also supports assembly language at the function level.

You will notice that both the C/C++ compiler and Pascal compiler support something called a Prefix File. If you enter a file name in here, the compiler automatically includes that file in every file it compiles. The prefix file is a great place to put a precompiled header, for example, or a header file that includes everything you need in your project. Then you don't have to include these files in every source code file you write.

FIGURE 7.7

The Pascal Compiler settings panel controls how this front-end compiler works.



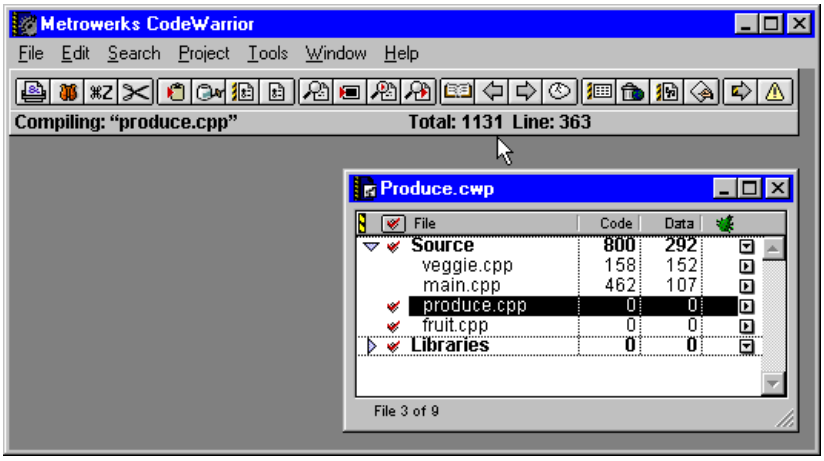
When working with the front-end (language) compilers, you have dozens of options. The task of setting these options individually for each project can be daunting. But there is an easy way to get the whole task accomplished painlessly. If you have one set of values you always use for new projects, set the CodeWarrior defaults to match your choices. Or, when you're all done setting up a project, you can always save it as custom stationery. You can then use that stationery as the basis for your next project.

Speeding Up a Compile

The CodeWarrior compilers are extremely fast. They can process tens of thousands of lines of code in a minute. As a compile progresses, the bottom part of the toolbar displays your progress. The file being compiled, the number of lines compiled, and the total number of lines compiled appear as shown in Figure 7.8.

FIGURE 7.8

The toolbar also shows you the compile-time progress.



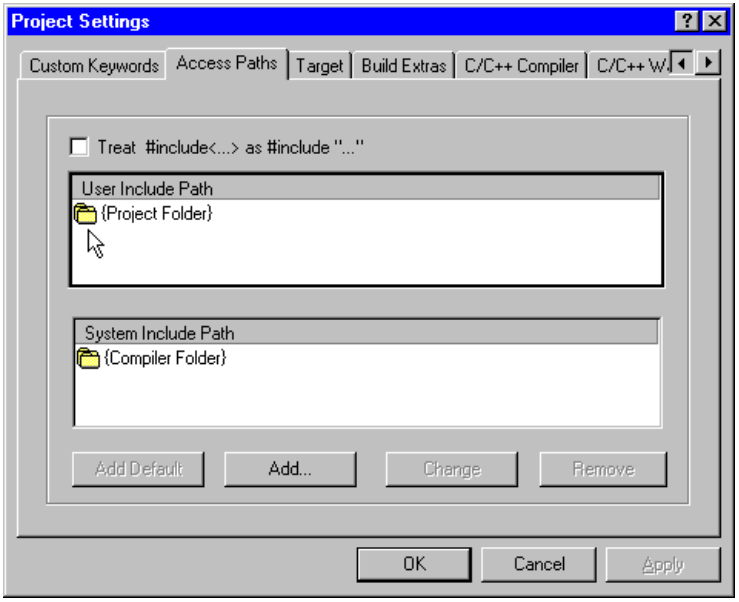
The first time you compile a file or a project takes a fair amount of time, because CodeWarrior must search along your access paths for all the files. After the first build, the search process is usually much faster.

However, even then, you may notice a substantial delay and a lot of disk thrashing before a file begins to compile. The compile itself is amazingly fast. It is the search for the file in advance of code generation that takes a lot of time. You can reduce this time by using advanced management techniques in the Access Paths settings panel. You met the Access Paths settings panel back in Chapter 3. Figure 7.9 shows the panel again.

As you know, this panel specifies the directories where CodeWarrior looks for files. By default, when you specify a folder in the Access Paths panel, CodeWarrior looks inside every folder in that directory. In other words, it performs a *recursive* search through the entire contents of a folder, including other folders. A recursive search can take a lot of time if the folder has lots of nested subfolders.

FIGURE 7.9

You can control recursive searching and search order in the Access Paths settings panel.



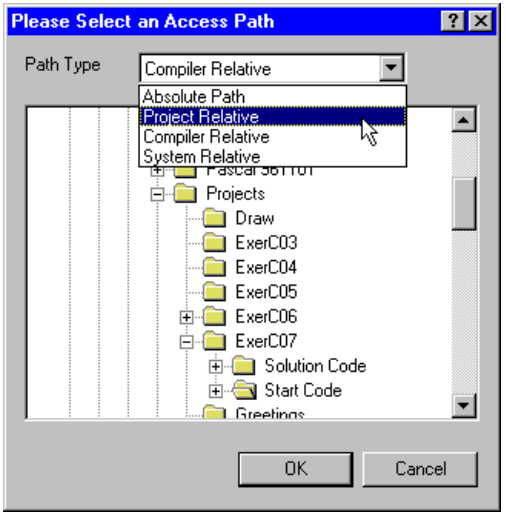
One of the best things you can do to speed up compile time is to turn off recursive searching. To the left of each path name in the Access Paths preference panel you'll find a folder icon. This icon indicates that CodeWarrior will use a recursive search through that folder. If you click that icon so that it goes away, CodeWarrior will not search recursively. It will just look for files in the folder, and not look inside subfolders. This makes the search *much* faster, but it also means that you must specify *every* folder that contains a file used in the project.

If you want to add a new path, decide whether you want to add a path to the User or System lists. After activating the correct list, simply click the Add button to display a dialog box that enables you to specify a new folder. Figure 7.10 shows this dialog box.

Use this dialog box to navigate to the folder you want to add to the access paths. You can add the folder in one of four ways:

- Absolute

FIGURE 7.10
You can add a new path and control how CodeWarrior uses the path using this dialog box.



- Project relative
- Compiler relative
- System relative

An absolute path represents the precise list of folders from the root directory all the way to the selected folder. The other three options tell CodeWarrior to save path information starting at either the project folder (which contains the project file), the compiler folder (which contains the CodeWarrior IDE), or the system folder. The system folder is the Windows folder on your main drive that contains your operating system software.

You can do more to reduce search time beyond specifying precise paths and turning off recursive searching. Even with precise paths, the search proceeds from top to bottom in the path list. So, the next best thing you can do to reduce search time is to put the most-used paths at the top of the list—and the least-used paths at the bottom. You can modify the list order by dragging items in the list to a new position, just as you would for files in the project window.

A third speed enhancement is to ensure that Treat #include <...> as #include "..." is *off*. When this option is off, the paths specified in the User Include Path are skipped for files you include with the <> syntax.

Finally, good hard drive management can help reduce the time it takes for CodeWarrior to find files. If you have "clean" folders without a lot of ancillary and nonproject related files in them, CodeWarrior has fewer files to look through.

TIP

If you use recursive searching, you can force CodeWarrior to skip a particular folder if the folder's name begins and ends with parentheses.

Taken together, the preceding tricks can cut your compile time by a factor of eight or ten over a worst-case scenario.

Disassembling Code

As the great chef likes to look in the pot from time to time to see how things are going, programmers can't resist looking at the actual processor instructions generated by the back-end compiler. This is called disassembling code.

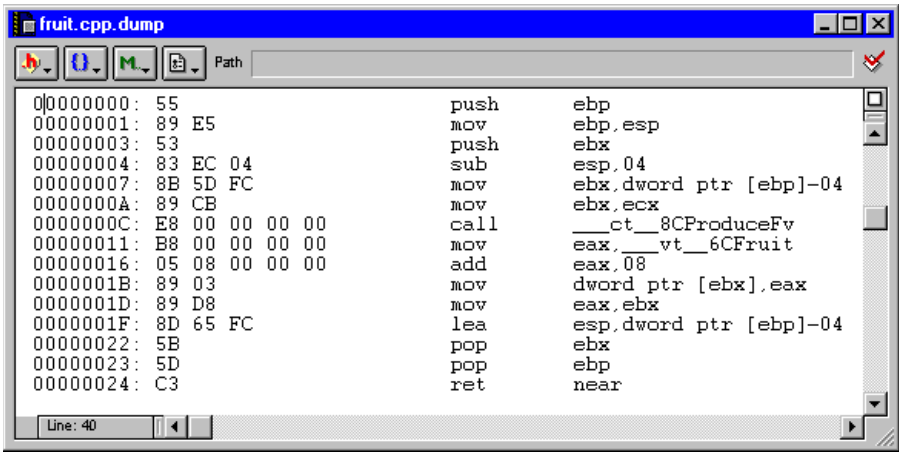
Looking at a disassembly can be very educational. For one thing, you can see how the choices you made among the various code generation options affected the final result. You can see exactly the kind of machine code the Metrowerks compiler generates for different kinds of source code constructs. You can even find the occasional subtle bug that eludes you at the source level. Studying disassembly gives you a real taste for the underlying machine-level instructions.

Disassembling code in CodeWarrior is trivially easy. It works just like compiling. All you have to do is select the file or files you're interested in and choose the Disassemble command from the Project menu. This command works on the selected file or files in the project win-

dow, or on the active editor window. When you issue this command, CodeWarrior compiles the file if necessary, and then the disassembly appears in an editor window. Figure 7.11 shows an example.

FIGURE 7.11

You can see the actual code generated by a CodeWarrior compiler with the Disassemble command.


fruit.cpp.dump
Path
00000000: 55 push ebp
00000001: 89 E5 mov ebp,esp
00000003: 53 push ebx
00000004: 83 EC 04 sub esp,04
00000007: 8B 5D FC mov ebx,dword ptr [ebp]-04
0000000A: 89 CB mov ebx,ecx
0000000C: E8 00 00 00 00 call ___ct__8CProduceFv
00000011: B8 00 00 00 00 mov eax,___vt__6CFruit
00000016: 05 08 00 00 00 add eax,08
0000001B: 89 03 mov dword ptr [ebx],eax
0000001D: 89 D8 mov eax,ebx
0000001F: 8D 65 FC lea esp,dword ptr [ebp]-04
00000022: 5B pop ebx
00000023: 5D pop ebp
00000024: C3 ret near
Line: 40

The editor window functionality is identical for this editor window as for any other. You can even jump to the assembly language functions! You can also save the resulting assembly-language code if you wish. Disassembly works no matter what the target. The resulting assembly language uses the instruction set for the project's specified target. For example, if you disassemble a Java .class file, you'll see JVM bytecodes. You can also see the assembly language code in the debugger, but that's a topic for the next chapter.

After you have a disassembly, you can open the original source file and compare the two side by side. You can see exactly how the compiler translated your source code. You can even use the assembly code as a model for your own assembly language functions.

Preprocessing a C/C++ File

In C/C++ code, you can include a variety of instructions to the preprocessor. The preprocessor prepares the code for the actual compiler. It reads a source file just before compiling, and performs some tasks on the code.

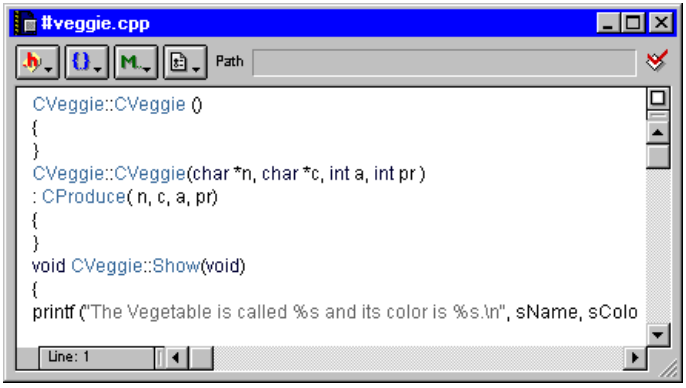
For example, one thing the preprocessor does is identify, interpret, and remove any statement that begins with a # sign. You might use a **#pragma** statement to set a compiler option directly in your code, rather than using the settings panel. This lets you modify settings on the fly for a small piece of code, and then return to some other setting. You might use a **#define** statement to create a macro. The preprocessor reads the macro and substitutes the actual code for the macro everywhere you used it in your code, thus "cleaning up" the file before the compiler sees it.

Unfortunately, you can't see what the preprocessor does during an actual compile—it works automatically. If you have complex or unusual macros, for example, they may not work exactly as you expect and can be the source of bugs. What you need in this case is the ability to see what the preprocessor does to the file *before* the compiler sees it.

You can do this with the Preprocess command in the Project menu. Like the disassembly command, select a file or files in the project window, or issue the command on the active editor window. Either way, the preprocessed file shows up in a new editor window, as shown in Figure 7.12.

As you can see, one of the things the preprocessor does is remove all white space and comments from a file. This can make the file a bit difficult to read. Nevertheless, this is the state of the raw data just before the compiler sees it, and that can be extremely useful to you in certain circumstances.

FIGURE 7.12
See the effect of the C/C++ preprocessor after issuing a Preprocess command.



Of course, this window has all the functionality of any editor window. You can save the contents of the window, split panes, jump to a function, add markers, and so forth.

Precompiling C/C++ Headers

In C/C++ code, many source files include a selection of header files that define function prototypes, constants, enumerations, structs, templates, and other code constructs used in the source file. In many cases, you have a general group of header files that get included all the time by all kinds of files.

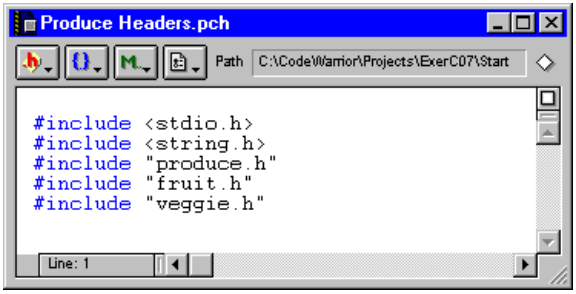
A good example of this is when you are programming with an application framework such as the Microsoft Foundation Classes (MFC) for Windows or PowerPlant for Mac OS. Similarly, the headers for the Windows SDK are numerous, complex, and included everywhere, but they take a long time to compile. This is particularly irksome when you are in the throes of a code-build-run-debug cycle. Every time you have to build, you sit idle while the compiler recompiles the headers, even though nothing has changed.

You can eliminate this problem by precompiling the headers once, and including the precompiled headers in the project. Using precompiled headers can greatly reduce the time it takes to build a project.

To make a precompiled header, first create a new header file that includes all the other headers you want to precompile, and save it with the naming extension .pch as shown in the window title in Figure 7.13. Then, when that file is the active window, choose Precompile from the Project menu. CodeWarrior precompiles the file and saves the precompiled header in the same folder as the source file, and with the same name (lacking the .pch extension).

FIGURE 7.13

You can create a precompiled header and save time whenever you build a project.

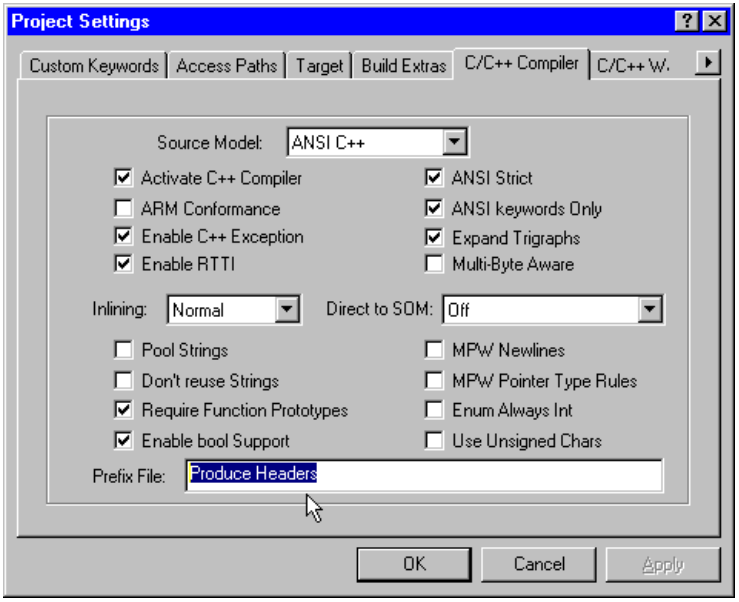


After you have built the precompiled header, you can get that file into your project in two ways. One way is to include it in the file that needs access to the information in the precompiled header. For example, if your precompiled header was named "PrecompHeader" you can include it with a line of code, **#include "PrecompHeader"**, just like you would for any other header file. You can also include it automatically via the prefix file setting in the C/C++ Language settings panel. (This is ideal for operating system headers such as the Win32 SDK.) Figure 7.14 illustrates this technique.

You can also add a precompiled header source file (the .pch file) right to the project. CodeWarrior interprets the .pch extension to mean

FIGURE 7.14

Use a precompiled header that you can include in your project for both speed and automatic updates.



that the file is a precompiled header. You create a header that starts with a single line of code that says

#pragma precompile_target "name" // use quote marks as shown

where "name" is the name of the file you want to create. Then include every header file you want in the precompiled header. It is usually best to make the precompiled header the first file in the project, because other source files probably rely on the information in this file.

Including the .pch file directly in a project provides two advantages. First, you don't have to manually precompile the file. CodeWarrior does it automatically when it encounters the file in the project. Second, if any of the included files changes, then CodeWarrior automatically updates the precompiled header without you having to lift a finger. Including a .pch file is the easy way to ensure that you get the benefits of increased speed and the reliability of automatic updates.

The only thing to remember is, even if you have the .pch file in the project window, you must still include the actual precompiled header file it produces by using the project prefix file technique or standard **#include** statements as described previously.

TIP

When working with the browser, .pch files are critical if you want to see classes included in the precompiled header (such as the MFC classes). The presence of the .pch file means the compiler actually sees the class declarations. As a result, the parser can generate browser database information for those classes. If you use the **#include** route and do not take the .pch route, the parser never sees the classes and the database does not include all the information you want.

Controlling a Link

Compiling the code gets you half-way to your goal. After all the object code has been created, you must put all that code together. That's what the linker does. The linker hooks up all the different pieces of object code into a final product. If function calls are made from one piece of object code to another, the linker makes the connection between them. The linker also analyzes all the code in any particular chunk, and strips out any that is unused—a process known as dead-stripping.

As we did for the compiler, let's see how to choose a linker, and how to issue a link command.

Choosing a Linker

Unlike front-end language and back-end target compilers, you don't have to specify which linker you want to use. Each target compiler has one associated linker. In effect, when you specify the target, you also specify the linker. No choice is involved beyond the target selection

you have already made, so the choice of linker is automatic. In fact, for some targets the choice is the linker!

For some targets you can also choose a post-linker. The post-linker performs additional work on the final code after it has been linked. The targets covered in this book do not require a post-linker, so we won't discuss this option any further except to say that if a post-linker is available, it appears as an option in the Post Linker pop-up menu in the Target settings panel. Like the target itself, choosing a post-linker is no more difficult than selecting an item from the list of available post-linkers. For example, if you are using CodeWarrior to program for the PlayStation game console, a post-linker is available that converts the final executable into a format suitable for downloading to the game console.

Issuing a Link Command

Three commands result in the object code for a project being linked into a final product. They are:

- Make
- Run
- Debug

Any of these three commands brings an entire project up to date by compiling all touched files, and then links the final object code into a finished product. Notice that if the code needs to be compiled, these commands both compile and link.

The linker will not run unless all the object code compiles without errors. However, the fact that the code compiles correctly is no guarantee that it will link correctly. For example, some piece of code may call a routine in a library that is not part of the project. As a result, the linker cannot connect to the called code, and a linker error results. We'll discuss how CodeWarrior handles errors in just a bit.

Assuming that no errors occur in the compiling and linking process, the Make command works for all kinds of projects. For example, if Win32x86 is your target, you can have three kinds of projects: an application, a library, or a dynamically linked library (DLL). A Make command will build any of the three. For Mac OS on the PowerPC chip you can make an application, library, shared library (equivalent of a DLL), or a code resource. Once again, Make is an appropriate command for all four possible project types.

The Make command creates the final code package, period. You'll see a new file in your project folder with the name of the final product that you specified in the linker settings. If the final product is an executable (an application, for example), you can also use the Run or Debug commands from the Project menu. The Run and Debug commands do everything that Make does, and then they launch the resulting executable automatically. Run just runs it normally. Debug launches the debugger and runs the application under debugger control.

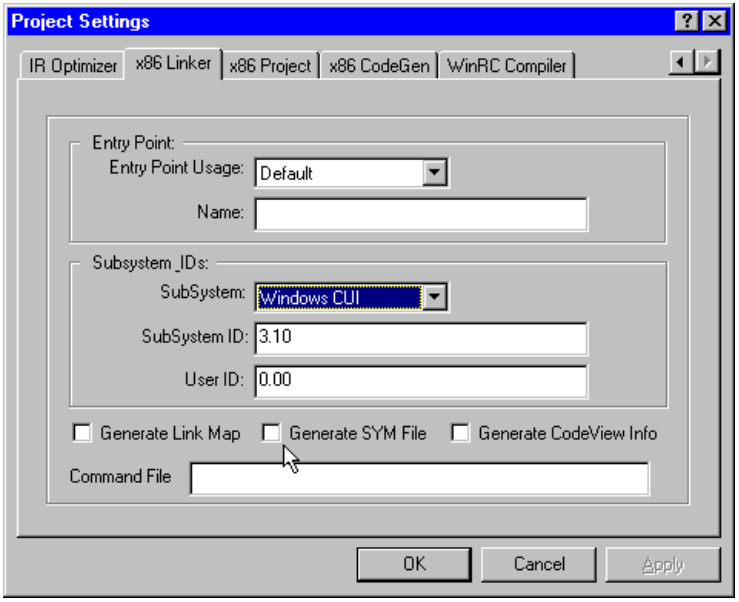
Setting Linker Options

As with code generation, each target linker has unique settings that let you control a variety of linker options. However, all the linkers have two options in common: You can tell them to generate a link map, and you can tell them to generate debug information.

For example, you'll find the linker options for the Win32x86 linker in the x86 Linker settings panel shown in Figure 7.15. The particular options of interest here relate to the link map and SYM file. The linkers for the Mac OS targets (both 68K and PowerPC) have similar options related to these two items.

The link map setting is near the bottom. If you turn this option on, the linker generates a file that contains information about the various functions in the code and how they are linked. This works for either

FIGURE 7.15
Use the x86 Linker settings panel to generate a link map and/or debugger information.



C/C++ or Pascal. The Pascal link map also includes information on dependencies between files, and the order in which files were compiled.

The Generate SYM File option tells the linker to create a SYM file for the project. The SYM file contains symbol information that the debugger needs in order to correlate the final object code with your source code. We'll discuss SYM files in some detail in the debugging chapter.

Beyond these two items, the linker options for various targets are quite different. We'll discuss them in subsequent chapters when we talk about writing code for Windows, Java, and Mac OS using CodeWarrior. For example, the x86 Linker can also generate debugging information in CodeView format.

Except for those target-specific features, you have covered the entire compiling and linking process in CodeWarrior with one exception—errors.

Cool Features

If the compiler doesn't like what it sees in your code, it generates errors and/or warnings. The linker can generate errors or warnings as well if it cannot hook up all the different pieces of object code.

The coolest thing about the compiling and linking process is how CodeWarrior handles errors and warnings. It uses the message window mechanism to give you real power and flexibility when dealing with mistakes. You've seen the message window concept twice already: once in the batch search results window, and again in the Symbol window for displaying multiple implementations of a symbol. You are about to meet it again.

Before you do, however, let's lay some groundwork for how you can control the kinds of errors and warnings you'll receive. Then we'll look at how to handle them.

Understanding Errors and Warnings

If the compiler encounters an error, you'll certainly hear about it. You cannot turn errors off, nor can you ignore them. If an error occurs while compiling code, the object code for that file cannot be built, and the project cannot be linked. End of story—until you fix the error.

The Metrowerks compilers can generate many different kinds of errors. The CodeWarrior documentation includes a manual devoted to them, the *CodeWarrior Error Reference*—150 pages of explanation about each and every possible error message that the compilers and linkers can generate. This manual is a particularly useful piece of documentation when you run into trouble. In many cases it explains the possible causes of the error, and also provides example code and solutions you can use to make the problem go away. This can be a real boon, especially to beginning programmers. After a while, though, you become familiar with the kinds of errors your particular coding style is likely to generate, and you also get to know how to fix them.

While no one likes to see mistakes, errors are particularly nice in one respect—you cannot ignore them. You know you have a problem, and unless you fix the problem you aren't going to go anywhere. There is no choice involved.

Unfortunately, other kinds of things can go wrong in code that do not cause errors. You can have perfectly legal language syntax, and still the code does exactly the wrong thing. A compiler, being a formal interpreter of legal syntax, can and should let all these perfectly legal—and perfectly wrong—statements pass without error.

However, a really friendly compiler should make every attempt to identify common language constructs that are legal, but usually wrong, and warn you about them. The Metrowerks compilers used in CodeWarrior are really friendly, and do this for you if you want them to.

Setting Compiler Warnings

A settings panel for each language controls how the compiler handles warnings. You can control whether to receive warnings, what warnings to receive, and whether to treat each warning as an error. Each language has its own set of warnings, because each language has its own peculiarities and syntax subtleties. (No warnings exist yet for the Java language.)

Figure 7.16 shows you the C/C++ Warnings panel. Figure 7.17 shows the Pascal Warnings panel.

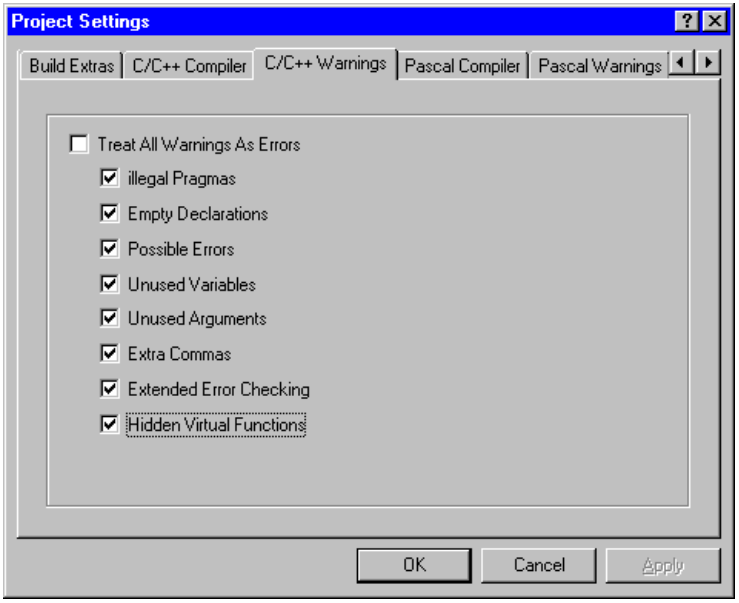
The nature of some of these errors is obvious, such as Unused Variables and Unused Arguments. Others are less obvious. For example, if the Possible Errors item in the C/C++ Warnings panel is on, the compiler warns you about:

- Unintended assignments in a logical expression, such as if **(a = b)** when you mean if **(a == b)**

- Unintended comparisons in a situation that normally requires an assignment, such as **a == 6;** when you mean **a = 6;**
- Semicolons after **while**, **if**, or **for** statements, such as for (**x = 0**, **x < 5, x++);**.

FIGURE 7.16

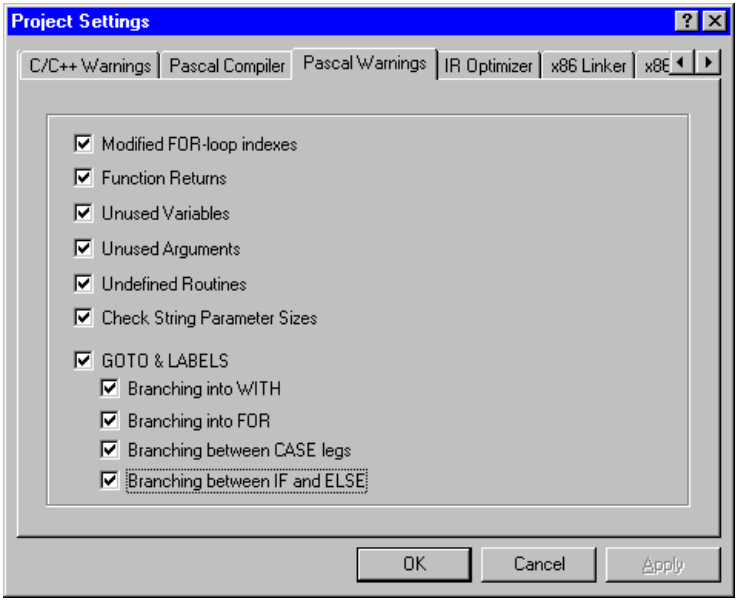
Use the C/C++ Warnings settings panel to activate or deactivate warnings for the C/C++ languages.



In each case, it is quite likely that the code is in error. In the first case, you are setting the variable **a** to be equal to **b**, instead of comparing the two. In the second case, you are testing to see if the variable **a** is already 6, instead of setting it to the value 6. And in the third situation the extra semicolon means that the loop does nothing.

The CodeWarrior documentation explains each of these possible warnings thoroughly. You'll find the C/C++ warnings discussed in the *C/C++/Asm Language Reference*, and the Pascal errors in the *Pascal Language Reference*.

FIGURE 7.17
Use the Pascal Warnings settings panel to activate or deactivate warnings for the Pascal language.



TIP

When working with warnings, the best policy is to turn them all on and leave them on. The kinds of code constructs flagged by the compilers as warnings are usually mistakes, even though they are legal according to strict syntax rules. However, you can get warnings when the code involved represents exactly what you want to do. In that case you can simply ignore the warning, or rewrite your code in such a way to eliminate the warning. If you follow the latter course, usually you end up writing cleaner, more easily understood code.

Now that you know how to control any warnings you get, let's see how CodeWarrior reports errors and warnings to you.

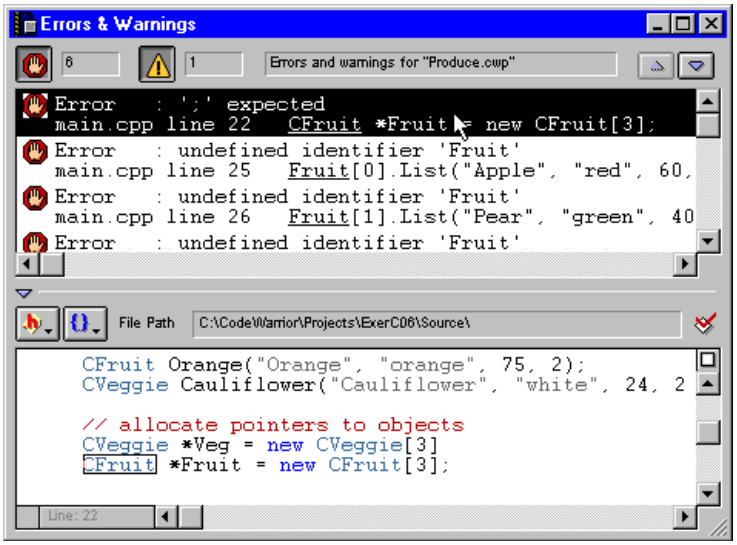
Understanding the Message Window

When an error or warning occurs in the compile or link process (assuming you have warnings turned on—and you should) the Errors & Warnings window appears. As you can see in Figure 7.18,

this window looks very much like the Search Results window you see after a batch search.

FIGURE 7.18

The Errors & Warnings window reports all errors and warnings to you from any compile or linking operation.


Errors & Warnings
6
1
Errors and warnings for "Produce.cwp"
Error : ';' expected
main.cpp line 22 CFruit *Fruit = new CFruit[3];
Error : undefined identifier 'Fruit'
main.cpp line 25 Fruit[0].List("Apple", "red", 60,
Error : undefined identifier 'Fruit'
main.cpp line 26 Fruit[1].List("Pear", "green", 40
Error : undefined identifier 'Fruit'
File Path
C:\CodeWarrior\Projects\ExerC06\Source\
CFruit Orange("Orange", "orange", 75, 2);
CVeggie Cauliflower("Cauliflower", "white", 24, 2
// allocate pointers to objects
CVeggie *Veg = new CVeggie[3]
CFruit *Fruit = new CFruit[3];
Line: 22

Buttons along the top of the window let you display or hide either errors or warnings or both. You can also see how many of each kind of message there are. An informational display lets you know what project is associated with this report. And the top right corner of the window has the Next and Previous buttons, just as in the Search Results window.

The top panel displays a list of all the errors or warnings, or both, in the order in which they occurred. You use the Next or Previous buttons to walk through the list, or select any item in the list as you wish. Whatever item you select, the associated code appears in the Source pane at the bottom of the window.

The Source pane displays the complete source file. It has many of the editor window features you have grown accustomed to, including:

- A function pop-up—to jump to any function in the file
- An interface pop-up—to open any included file
- A file path display—to see where the file is on disk
- A go-to-line button—to jump to any line in the file
- A file changed icon—to show whether the file has changed
- An appearance button—to use small icons for the pop-up menus
- A resize bar—to resize the panes
- A disclosure triangle—to show or hide the Source pane

Just like the Search Results window that displayed hits from a batch search, this window displays "hits" from the compiler. The only difference is, this list of hits is compiler and linker errors and warnings. Other than that, the functionality is the same.

Dealing with Errors

The Errors & Warnings window makes dealing with errors very easy, because you are already master of the tool and the tool is especially slick. You know that you don't have to open the file associated with the error, or remember what file contains the problematic code. Just select the error, and there it is in the Source pane. If for some reason you want to open the file in a separate editor window, just double-click an item in the list. The file opens in a full editor window.

However, you won't really need the editor window most of the time. You can see each error right in the Errors & Warnings window, resolve the error in place, and move on. You can close this window and come back to it at a later time if you wish. Use the Errors & Warnings Window item in the Window menu to show this window. The contents will remain intact until you do another compile.

Once again, the message window performs its magic. File boundaries and locations become meaningless. You are presented with what

you need to know right at your fingertips. Don't forget, if you have the browser on, the context pop-up menu can take you to lots of other places in your code as well. If you need to see an interface file, a function declaration, a variable definition, a constant value, or implementations of a function, all these destinations are just a click away.

TIP

Two very cool things are going on here. One is the error window itself, which—although now familiar to you—is still absolutely marvelous. The second is, you didn't have to learn a whole new interface! The same window you use for batch searches also works for compiler errors. Now that's cool.

As you could with the other incarnations of the message window, you can save the contents of this window at any time. To save the error and warning list, make sure that's the active pane and choose the Save A Copy As command from the File menu. To save the source file, make the Source pane the active pane and choose the Save command. You can print either pane as well if you'd like to get a list of your errors on paper or print the source file.

This brings us to the end of the discussion about how compiling and linking work, and how you handle errors. Let's see how all this comes together with some real code.

Hands-On Exercises

In this exercise you'll use the same, simple project you explored in the previous chapter—the Produce project. In Chapter 6 you explored some of that code with the browser, and in this exercise you will compile and build that project into a finished executable. It's not a Windows application, it is a "console" application using ANSI code, but it runs just the same. We'll discuss Windows programming with CodeWarrior in subsequent chapters.

1. Launch CodeWarrior and open the project. If CodeWarrior is not already running, launch CodeWarrior, and then open the produce .cwp project file from the Start Code folder, which is inside the exercise folder for this chapter, Exer07.

Make sure you use the start code from the Exer07 folder, and not the code from Chapter 6 or the solution code for this exercise. I have introduced a few diabolical errors in the code that you will discover and correct in subsequent steps.

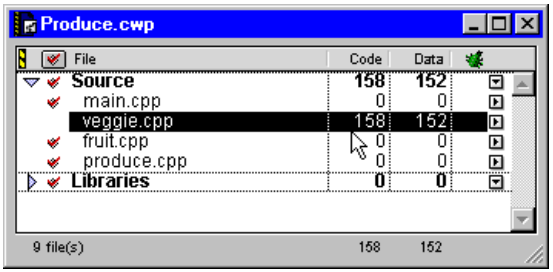
2. Set the compiler warnings. As you know, if the compiler encounters an error you will be notified. In this step you set project-related options to ensure you get warnings if the compiler encounters potential problems in your code. To begin, go to the project settings (not the global preferences) and examine the C++ Compiler panel. This is already set to ANSI C++, which is fine for this project. If you wanted to modify how the compiler works, you would modify these settings. For this exercise, you should leave them unchanged.

Next, go to the C/C++ Warnings panel. In this panel, turn on all the warnings as shown in previously in Figure 7.16. You do not have to turn on the Treat All Warnings as Errors item; you can treat them as what they are—warnings. When you are through, click the OK button to accept the changes.

3. Compile a single file. In this step you generate code for a single file. None of the project files have been compiled yet, but just do one file at this time. You'll do the rest in the next step.

Select the veggie.cpp file in the project window and choose Compile (Ctrl+K) from the Project menu. CodeWarrior generates the code for this file. As the file compiles, keep an eye on the toolbar where progress is displayed. When the compile is done, the Code and Data sections of the project window will be updated to reflect the object code created for this file. The check mark to the left of the file name disappears, so you know this file is up to date. Figure 7.19 shows what the project window looks like after the compile.

FIGURE 7.19
Compiling generates the object code for a file.



4. Bring the entire project up to date. In this step you compile all the remaining files in the project. To do this you issue a single command—Bring Up To Date (Ctrl+U) in the Project menu. Once again, watch the progress in the toolbar as the compile proceeds. However, this time you're going to run into problems. The code contains errors, and when the compiler finds the first problem the Errors & Warnings window will appear.

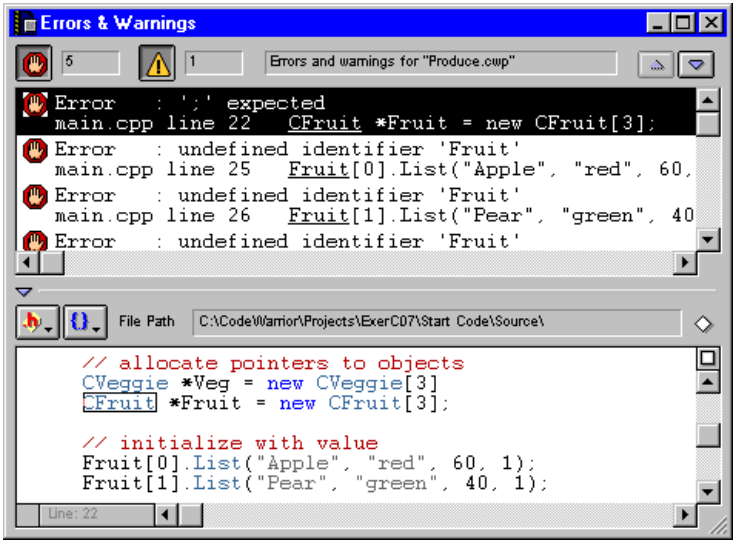
However, the compiler does not stop working at the first sign of trouble. It continues to read your code despite the error, so that it can present you with a complete list of problems. The list of errors in the window grows as the compiler continues working on the project, and the final result should look like Figure 7.20.

Look at the project window and you'll see that all files successfully compiled except the main.cpp file. All the problems are in that file—five errors and one warning.

5. Use the error window to resolve problems. In this step you fix the problems with the code. Because the compiler continues work even after it encounters an error, many subsequent errors can be spurious—that is, caused because an earlier error confused the compiler. In this step all of the errors will go away when you fix the first one—one error is real, and four are spurious. (The warning is real as well, but you'll take care of that a little later in this step.)

FIGURE 7.20

The Errors & Warnings window appears listing every error and warning encountered by the compiler.


Errors & Warnings
5
1
Errors and warnings for "Produce.cwp"
Error : ';' expected
main.cpp line 22 CFruit *Fruit = new CFruit[3];
Error : undefined identifier 'Fruit'
main.cpp line 25 Fruit[0].List("Apple", "red", 60,
Error : undefined identifier 'Fruit'
main.cpp line 26 Fruit[1].List("Pear", "green", 40
Error : undefined identifier 'Fruit'
File Path
C:\CodeWarrior\Projects\ExerC07\Start Code\Source\
// allocate pointers to objects
CVeggie *Veg = new CVeggie[3]
CFruit *Fruit = new CFruit[3];
// initialize with value
Fruit[0].List("Apple", "red", 60, 1);
Fruit[1].List("Pear", "green", 40, 1);
Line: 22

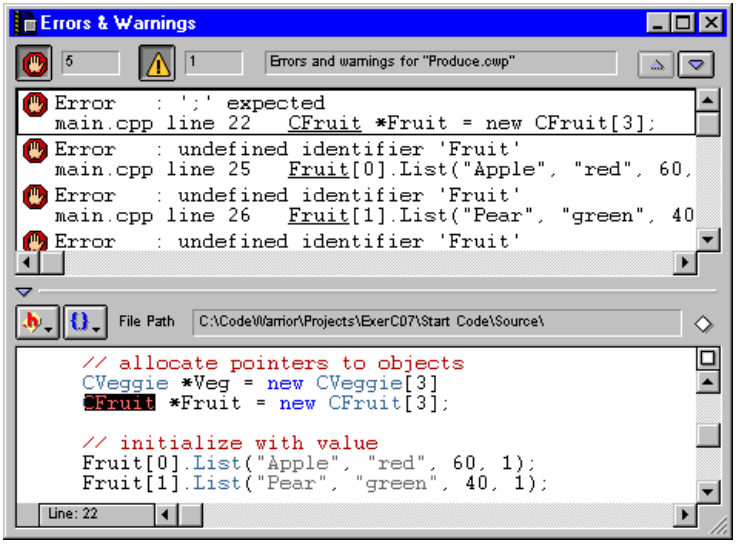
In this case, the first error message is selected, and it says that a semicolon is expected. The message provides the line number where the problem is located, but you can press the Tab key to jump to the location of the error in the Source pane. Figure 7.21 shows you what this looks like. Notice that the class name CFruit is selected, because that's the point at which the compiler had a problem. However, if you look at that line of code, it already has a semicolon at the end. It is the line *before* this statement that is missing the semicolon.

Whenever you encounter an error, the actual problem is always at or *before* the highlighted text. Sometimes the error window reports the wrong line because the compiler doesn't run into problems until a subsequent line of code is compiled. The missing semicolon is a classic example. A missing closing brace at the end of a function causes the same kind of problem. The actual error *never* occurs after the stated line, because the compiler hasn't seen that code yet.

To fix this problem, type a semicolon at the end of the previous line of code that allocates some CVeggie objects. That's where the real problem is. When you are through, that line of code should look like this:

FIGURE 7.21

After you select an error, press the Tab key to jump right to the error in the code.


Errors & Warnings
5
1
Errors and warnings for "Produce.cwp"
Error : ';' expected
main.cpp line 22 CFruit *Fruit = new CFruit[3];
Error : undefined identifier 'Fruit'
main.cpp line 25 Fruit[0].List("Apple", "red", 60,
Error : undefined identifier 'Fruit'
main.cpp line 26 Fruit[1].List("Pear", "green", 40
Error : undefined identifier 'Fruit'
File Path
C:\CodeWarrior\Projects\ExerC07\Start Code\Source\
// allocate pointers to objects
CVeggie *Veg = new CVeggie[3]
CFruit *Fruit = new CFruit[3];
// initialize with value
Fruit[0].List("Apple", "red", 60, 1);
Fruit[1].List("Pear", "green", 40, 1);
Line: 22

TIP

In some cases, the error can even be in another file! For example, if you forget to put the semicolon at the end of a class declaration in a header file, that can cause an error to be reported in a source file.

CVeggie *Veg = new CVeggie[3]; // semicolon at end of this line

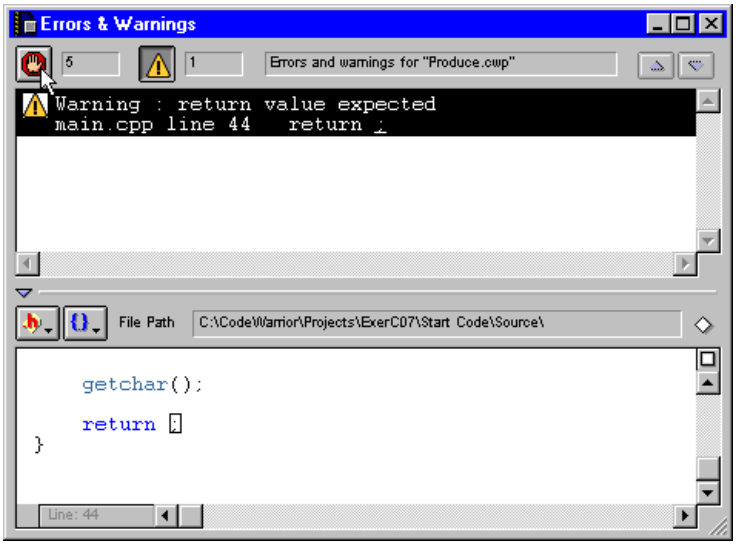
The remaining four errors all relate to an unknown identifier, the symbol "Fruit." These are spurious errors caused by the problem you just fixed. Because no semicolon appeared at the end of the "Veggie" line, the compiler couldn't understand the "Fruit" line—ergo, any time you try to use the local variable **Fruit** the compiler gets confused and coughs up an error. In effect, fixing error number one eliminated all the errors in the project. Let's hope all your debugging is this easy!

Before we go on, however, one warning remains to be resolved. You can find this warning in the errors and warnings list using several techniques. For example, you can click in the list and scroll until you see the warning icon to the left of a message, or you can turn off the errors by clicking the error button at the top left of the window. This

second approach leaves just warning messages displayed, as shown in Figure 7.22. You can also use the Next Error button at the top right of the window to walk through the list of errors and warnings.

FIGURE 7.22

You can display either errors or warnings, or both as you prefer.



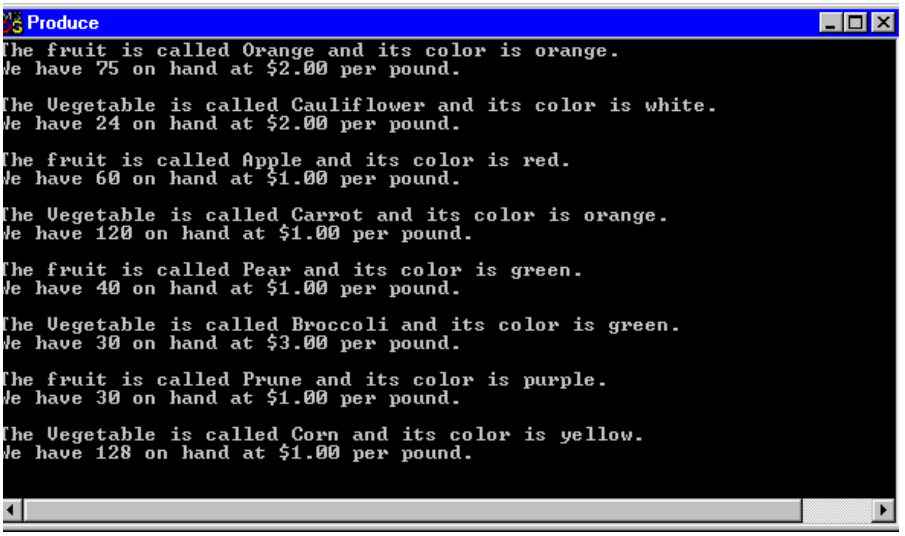
When you have the warning selected, press the Tab key to go right to the error in the code. Here, the problem is that the function **main()** should return a value, and it doesn't. By convention, it should return the value zero. Change the line of code to read

return 0;

and the warning will go away.

When you are all through, close the error window. You should see an alert message that unsaved changes exist in main.cpp. Aha! Although you have been working in the error window, CodeWarrior was accessing the actual file main.cpp. You never had to open the file—CodeWarrior took care of that for you, and is also taking care of forgetfulness here. Save the changes you made by clicking the Save button.

FIGURE 7.23
The grocer's report lists available fruits and vegetables.


Produce
The fruit is called Orange and its color is orange.
We have 75 on hand at $2.00 per pound.
The Vegetable is called Cauliflower and its color is white.
We have 24 on hand at $2.00 per pound.
The fruit is called Apple and its color is red.
We have 60 on hand at $1.00 per pound.
The Vegetable is called Carrot and its color is orange.
We have 120 on hand at $1.00 per pound.
The fruit is called Pear and its color is green.
We have 40 on hand at $1.00 per pound.
The Vegetable is called Broccoli and its color is green.
We have 30 on hand at $3.00 per pound.
The fruit is called Prune and its color is purple.
We have 30 on hand at $1.00 per pound.
The Vegetable is called Corn and its color is yellow.
We have 128 on hand at $1.00 per pound.

6. Make the project. In this step you build the final application. Having corrected all errors and warnings, you could bring the project up to date again, but that doesn't link the final executable. To build the application, issue a Make command (Ctrl+M) from the Project menu. The Make command both updates the entire project *and* links the final executable. When you're all through, the sign of success is the absence of failure. If the error window doesn't appear, and all the files in the project window have object code, you're done!

7. Run the executable. Just choose Run from the Project menu to launch the application, and there it is. Figure 7.23 shows the final output. Press the Enter key when you're done observing to exit from the console application.

TIP

Actually, Run brings the project up to date, links it, *and* runs it. However, the Run command is unavailable for projects that do not create executables, and in those cases you must use Make to link the final code.

Congratulations. It took us a while, but you have built a CodeWarrior project into a finished application. The remaining steps in this exercise are optional, but worthwhile. In the rest of the exercise you build and use a precompiled header—a valuable technique when you work with large projects.

8. Create a precompiled header. Now that you have built a simple application, in this step you practice a way to build a C/C++ application more efficiently—the precompiled header. It won't make a noticeable difference for a simple project like this, but it will make a world of difference when it comes time to build full-blown Windows applications.

Return to the CodeWarrior environment, where your project awaits further work. Create a new source file, and type the following lines of code that represent all the headers used in this project. You could copy or drag the text from existing source files if you wish.

```
#include <stdio.h>
#include <string.h>
#include "produce.h"
#include "fruit.h"
#include "veggie.h"
```

Then save the file with the name Produce Headers.pch (save it in the Start Code folder alongside the project file) and close the new .pch file.

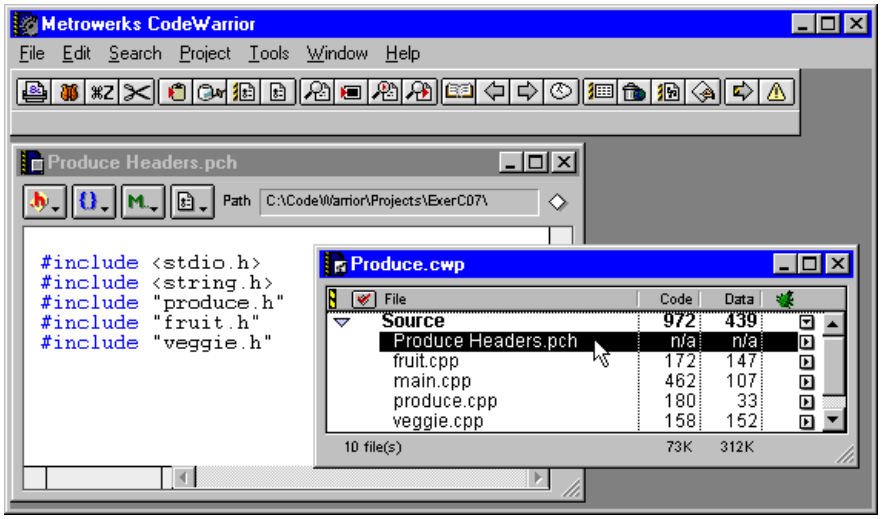
WARNING

Sometimes the IDE is just a bit too helpful. When you save the .pch file, the IDE might add a .txt extension for you, because it thinks you're trying to save a text file. If it does, you can manually remove the offending .txt extension from the file, and continue with the rest of the step.

Next, choose the Add Files item from the Project menu, navigate to the new .pch file, and add it to the project. Figure 7.24 shows the project window with the file added and selected. The actual file itself is also open to show you the contents.

FIGURE 7.24

The precompiled header file is ready to be precompiled.

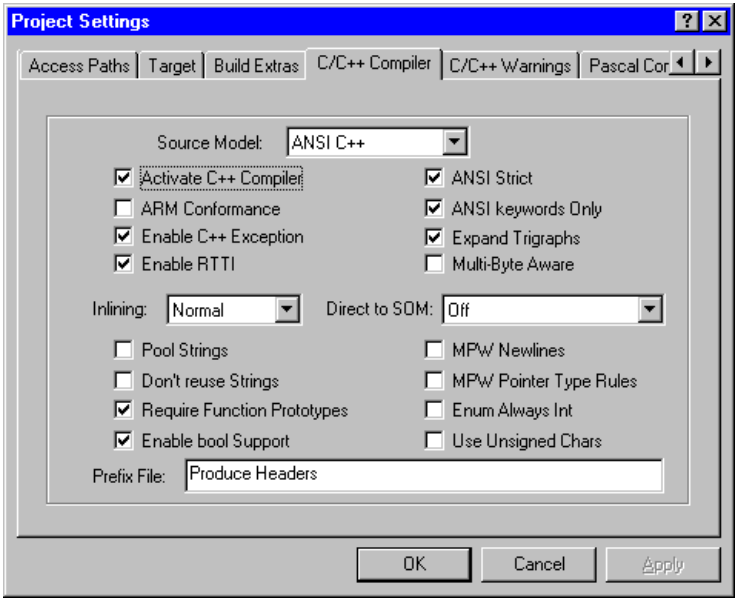


With the new file selected, choose Precompile from the Project menu, and CodeWarrior precompiles all the headers. It creates a file named Produce Headers—the same name as the .pch file without the extension—and puts it in the same folder as the original .pch file. You can go to the Windows desktop and take a look if you'd like to confirm that the file exists at the proper location. You have successfully created a precompiled header. Now it's time to use it.

9. Use a precompiled header. The simplest way to use a precompiled header is to set the project prefix file. Go to the project settings and open up the C/C++ Compiler panel. Near the bottom of the panel, enter the name of the precompiled header file—Produce Headers. Figure 7.25 shows you the panel with the prefix file set.

Because you now have the headers automatically included in all your project files via the precompiled header, *all* the include statements anywhere in your own project files can be removed. Use the

FIGURE 7.25
Using the prefix file is a great way to include your precompiled header file.



multi-file search capabilities of CodeWarrior to look for the **#include** statement, and comment them out. I'll leave the details up to you.

WARNING

Don't use the Replace All feature—you might unintentionally find #include statements inside some of the standard library headers such as stdio.h or string.h. You don't want to remove those. You only want to eliminate #include statements from your own files.

When you are through, make the project again. If it does not compile successfully, revisit the steps above and ensure that you have followed them carefully. Study the error messages, figure out what went wrong, and fix each error. You have the necessary knowledge and skill. If all

else fails, check your work against the solution code to see where it differs.

10. Run the executable. When the project builds successfully, run it and enjoy the fruits (and vegetables) of your labor.

Other aspects to compiling and linking that this exercise has not touched upon include viewing disassembly. Feel free to go back through the discussion in this chapter and explore other aspects of CodeWarrior. When you are all through, you can exit from CodeWarrior if you wish.

Summary

This chapter took you right to the heart of CodeWarrior—how to use the IDE to manage and control compiling and linking. As you have seen, compiling and linking is virtually automatic. However, if you want to get down into the inner workings of the machine you have complete freedom to do so. You are familiar with the plug-in architecture of front-end language compilers and back-end target compilers. You can pick a target and pick a language. You also know all about the various options for controlling the front-end compilers and language warnings. Despite the complexities presented, you also know that by using the project stationery concept you can greatly simplify all the setup work required to get a controlled compile operation under way.

You also became familiar with the way CodeWarrior handles compiler and linker errors and warnings. In fact, you mastered it without ever seeing it, because the Errors & Warnings window is just one more incarnation of the CodeWarrior message window. The mechanism itself, and the fact that it matches the rest of the CodeWarrior interface, are both cool features.

With this chapter you have reached a milestone. You can now write, compile, and link a complete project using CodeWarrior. Congratulations, and well done!

Next we undertake a survey of the last core tool in the CodeWarrior IDE, the debugger. No code is ever perfect the first time, and no development environment is complete without a good debugger. In the next chapter we take a good, long look at MW Debug, the CodeWarrior debugger.

CHAPTER

EIGHT

8

Debugging Your Code

- Understanding the Debugger
- Turning On the Debugger
- Understanding the Debugger Windows
- Mastering Basic Debugging
- Cool Features
- Hands-On Exercises

The CodeWarrior debugger is a stand-alone application that you can launch and minimally control from within the CodeWarrior IDE. The official name of the CodeWarrior debugger is MW Debug. However, for simplicity let's just call it the debugger.

In this chapter you'll learn how to set up a project for debugging, how to launch the debugger, and how to use the debugger to study your source code while it runs. You know that debugging is not as much fun as playing *Doom* or *Marathon*, but it is a vital and necessary part of code development. This chapter is devoted to exploring the multilanguage, multitarget debugging tools that are part of CodeWarrior.

Understanding the Debugger

The CodeWarrior debugger is a source-level debugger—it displays your original source code to you, line by line, as the program actually runs on the host computer. You can then study the behavior of your code as it actually works.

What is really running on the machine is not your source code, however, but the final object code that has been compiled and linked by CodeWarrior. These machine instructions have a correspondence to your source code, but they aren't the same thing. The debugger requires special symbol-based information—called *symbolics*—that tells it (among other things) how each instruction in the object code relates to the original source code. Based on that correspondence, when the debugger encounters a particular line of machine code, it can locate and display the proper line of source code to go along with it. In this chapter I'll use the terms *debug information* and *symbolics* as synonyms for this data.

The format of the information stored in the symbolics can vary. CodeWarrior currently understands several completely different formats used for different targets, including both CodeView and SYM. The CodeView format is common for Windows programming. The SYM standard is common on the Mac OS, and can be used on Windows as well.

The compiler and linker generate all the debugging information stored in the symbolics, but where that information is stored depends on whether you are using CodeView or SYM, or working with Java code. CodeView information is stored inside the actual .EXE file or DLL along with the machine code. SYM data is stored in a separate file that uses the naming extension .SYM. If you are working with Java code, the symbolics information is stored inside the class file. I will call a file that contains this information a *symbolics file*, regardless of whether it is a SYM, class, or .EXE file.

You decide whether to generate debugging information and which format to use by modifying project settings. Understanding this fundamental link between source code and the debugger is important to mastering how the tools work. This link affects how you control your project, and how you run the debugger. Of course, the debugger does much more than just display your source code, but that's what this chapter is all about.

Turning On the Debugger

Because the debugger is a stand-alone application that can be controlled from within the CodeWarrior IDE, you can control it either independently or via the IDE. Let's look at both approaches, because you are likely to use them both when you use CodeWarrior. However, let's start our discussion from the IDE, because that's where we have been all along in this book. And that's where you must do the necessary setup work so the debugger gets the required symbolics data.

The Thought behind the Debugger

As with everything else in the CodeWarrior environment, a lot of thought has gone into the debugger's design. The fundamental principle is that the same debugging interface should work for all your code. It shouldn't matter what language you use, what target you compile for, or whether your project is an executable application or a library. It shouldn't even matter what processor you are running on. All of the complexities required to work on multiple hosts, using multiple languages, for multiple targets should be and are hidden from you.

This approach gives you some real advantages. The most obvious benefit is the time you save because you don't have to learn new tools. Once you master the CodeWarrior debugger, you're ready to go—no matter what language, target, or processor is involved. You have a second, less obvious gain as well, but it is much more significant in the long run. Bear with me as we explore what this benefit is.

There have been and still are arbitrary but real boundaries between computer platforms, operating systems, and programming languages. Throughout this book you have seen how the CodeWarrior architecture and the tools break down old and artificial boundaries. Even the way we organize our code—from a file-based perspective—is a hindrance. Again, CodeWarrior makes real progress in minimizing the effects of these barriers. The message window and the browser make the old-fashioned file-based structure much less of a hindrance.

These kinds of boundaries get in the way of productivity. When barriers come down, new ideas flow. The trend now is toward cross-platform programming. Hardware is reaching the market that lets you pick the operating system you want to use. A new language and virtual platform, Java, is the latest attempt at minimizing cross-platform programming issues. Let's use Java—both the language and the target—as a real case study.

continued ➠

The Thought behind the Debugger

Java is an object-oriented programming language. In theory you should be able to program in Java for any target. However, for now at least, if you write code in the Java language you're limited to writing code for the Java Virtual Machine—the Java target. The idea behind the JVM and the accompanying class libraries is platform-neutrality. Using the JVM approach, work required to translate JVM code to native code for a particular processor is transferred from you, the programmer, to the tool developer or operating system engineer. For example, Metrowerks provides a Mac OS Java runtime that interprets the JVM calls. Major OS providers such as Microsoft and Apple Computer have already announced plans to build the necessary interpreters and just-in-time compilers right into their respective operating systems, making stand-alone runtime interpreters obsolete.

As the boundaries between platforms disappear, so to do the boundaries between languages. Java as a programming language has some very nice features, but you can perform some tricks in C and C++ that you simply cannot do in Java. So Java programmers find themselves writing and calling C functions.

If you write multilanguage code like this, you have a problem. How do you debug this code? If you're working in your Java environment, it doesn't understand C/C++. If you're working in your C/C++ environment, it doesn't understand Java. More to the point, you know that the only way you can properly debug your code is to have it all running as an integrated whole so you can see it work.

That's where the CodeWarrior debugger comes in, and this is the second major benefit to you of the thought behind the debugger. It may be subtle, but it is quite real. Being a multilanguage debugger, you can effortlessly step from one language to another in the CodeWarrior debugger. If your Java code calls a native C function, you can step right

continued ➠

The Thought behind the Debugger

into that function without blinking an eye. Likewise, if you call a C library routine from within your Pascal code, no problem. The transition is completely invisible to you as the programmer and user.

And isn't that how it should be? The purpose of a set of tools is to give you the means to an end. The tools should not get in the way. If you want to program in multiple languages, your tools should help you. That's exactly what CodeWarrior does.

Setting Up Debugging Information

The IDE does two things with the debugger. It can:

- Generate debugging information
- Launch the debugger

The first of these two items—generating debugging information—is the sole province of the IDE. The compiler and linker—as directed by the project manager in the IDE—create this debugging information. There is no other source. Consequently, you cannot debug any code unless you have first set it up properly and instructed the IDE that you want debugging information. The simplest way to do this is to choose the Enable Debugger item from the IDE project menu, as shown in Figure 8.1.

When you issue this command, the IDE sets a few options in the project settings. If these are not already set properly, you may see a dialog box warning you that certain settings must be changed. Figure 8.2 shows you this dialog box. Simply click OK and debugging is enabled.

FIGURE 8.1

Choosing the Enable Debugger item does all the necessary setup in the IDE.

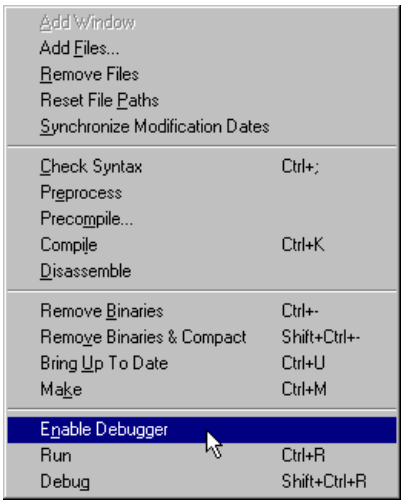
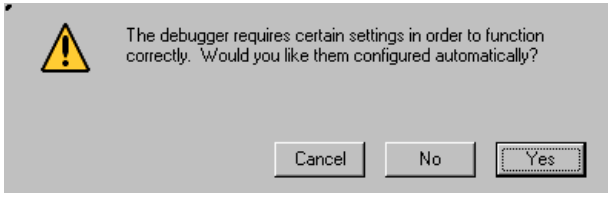


FIGURE 8.2

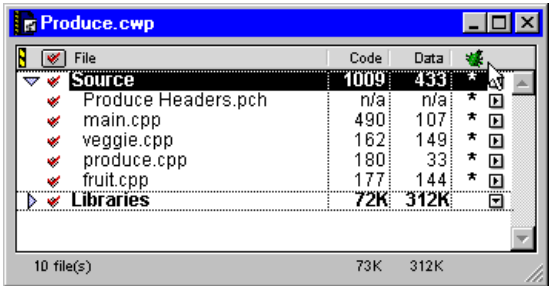
Before changing project debugging settings, the IDE warns you and gives you the opportunity to cancel.


The debugger requires certain settings in order to function correctly. Would you like them configured automatically?
Cancel
No
Yes

Enable Debugging also turns on debugging for all your source files automatically—you'll see marks appear in the debug column of the project window for every debuggable file. Debug information is generated from your source code on a file-by-file basis. Typically, you want debugging information for *all* your source files. Back in Chapter 3 you met the debugging column in the project window, shown in Figure 8.3 here. You know that you can turn on debugging information by clicking in that column for any individual file or group. You can turn debugging on or off for every file by Alt+clicking for any file or group.

FIGURE 8.3

The debug column lets you control which files have debug information created for them.



So much for the automatic approach. You can also do the setup work manually if you prefer, and in some cases it is advisable that you do. You have two completely different ways of creating debug information—CodeView or SYM—and you may prefer one or the other, or both. You can control debug information manually through the Project Settings. Two panels are involved: the x86 CodeGen panel and the x86 Linker panel—the back-end compiler and linker.

Figure 8.4 shows you the x86 CodeGen panel. The two options are at the bottom of the panel, one for CodeView format and the other for SYM format. Turn on one or the other or both of the debug information formats you're interested in. You can generate both formats simultaneously if you wish.

The x86 Linker panel shown in Figure 8.5 tells the linker how to output the debug information generated by the compiler. Once again you have two options—one to Generate CodeView Info and the other to Generate SYM file. Use the option that corresponds to the setting in the x86 CodeGen panel. In other words, if you select CodeView format in the x86 CodeGen panel, select Generate CodeView Info in the x86Linker panel. You cannot mix format and output—for example, you can't store CodeView-formatted debug information in a SYM file. You must match your options in these two panels. You can, however, output both kinds of data simultaneously if you wish.

FIGURE 8.4

Use the x86 CodeGen panel to tell the compiler what format to use for symbolics.

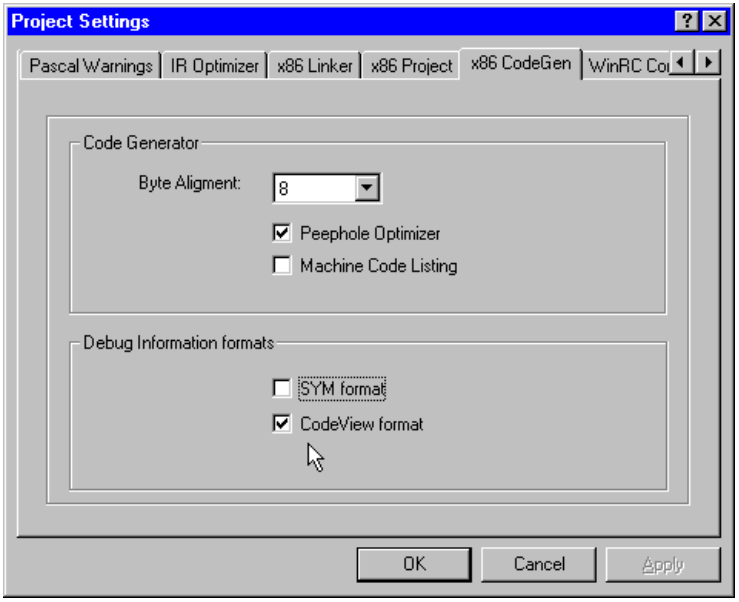
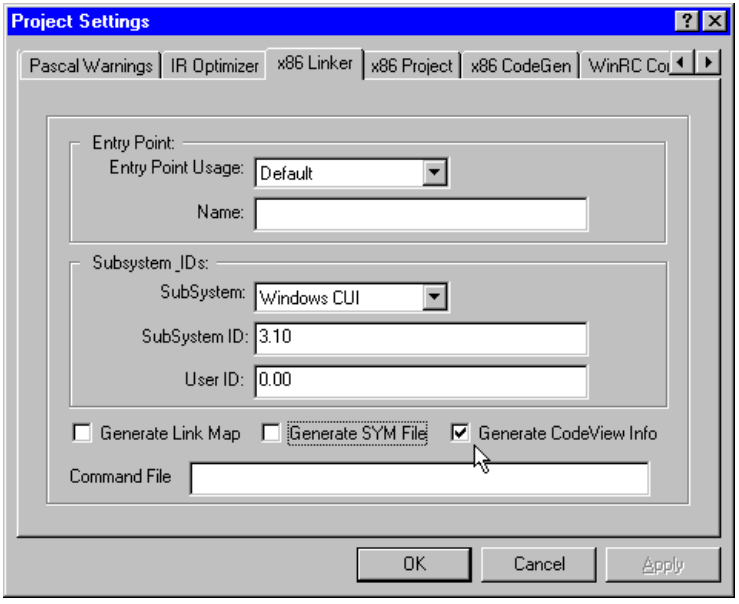


FIGURE 8.5

Use the x86 Linker panel to tell the linker how to output the symbolics.



NOTE

Choosing Enable Debugger turns on the CodeView options in the x86 CodeGen and x86Linker panels for the project, and marks all source files for debugging. If you want to use SYM format, you have to turn it on manually.

Having enabled the debugger and told the project manager that you want debug information for your source files, you then link your project using the Make command (or the Debug command). As it creates the final executable or library, the compiler generates the necessary data and the linker outputs the data for the debugger—either installing it in the final code (for CodeView) or creating a SYM file. The SYM file is placed with your project file and named the same as your final program, but with the .SYM naming extension added to the project name. For example, a project that generated MyApp.EXE would have a SYM file named MyApp.EXE.SYM.

The process of creating debug information does *not* automatically launch the debugger. It simply creates the data that the debugger needs. Symbolics data is generated by a Make operation—that is, when you link the project into a final software product, be it executable or library.

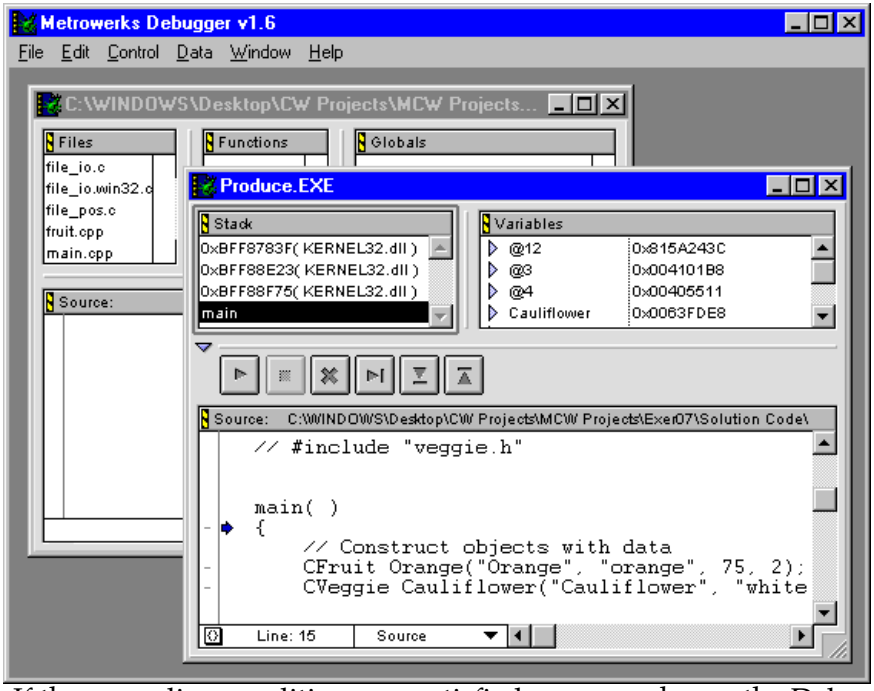
Launching the Debugger

As you know, from within the IDE you can link a project with one of three commands from the Project menu:

- Make
- Run
- Debug

Each of these commands causes the project to be linked. As you might guess, the only one of these three commands that launches the debugger is the Debug command. This command is available only if debugging is enabled, and only if you are creating an executable.

FIGURE 8.6
When launched from the IDE, the debugger appears with your program launched and ready to go.



If the preceding conditions are satisfied, you can choose the Debug command so that CodeWarrior launches the debugger. You'll see the standard dialog box to open the symbolics file. At that time, navigate to the symbolics file (which may be the executable, a SYM file, or a Java class file) and open it. You are then ready to debug.

TIP

Because the CodeWarrior debugger understands CodeView format, used by Visual C++, you can build an executable in that environment and debug it in CodeWarrior. Of course, you must link it using CodeView symbols in Visual C++, but if you have done that, you can just launch the CodeWarrior debugger and open the Visual C++-created executable.

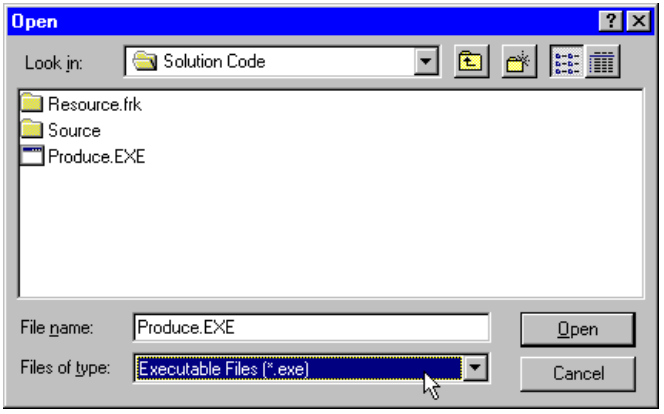
After you open the symbolics file, two windows appear, as shown in Figure 8.6. These are the debugger's Program and Source Browser windows. We'll discuss them in detail in just a bit. (Actually a log window will also show up, but we'll discuss that later on as well.)

Before we discuss these windows, however, what do you do if your project is not an executable? How do you get started debugging a library? We'll discuss the details later on, but the quick answer is, you launch the debugger as a separate application. Simply launch the debugger as you would any other program.

When you launch the debugger directly, the first thing that happens is it asks you for a file. You use the standard file dialog box shown in Figure 8.7 to locate and open the symbolics file for the project you're debugging. This might be the .EXE or .DLL file (if you're using CodeView), a SYM file, or a Java class file. You can filter visible files with the Files of Type combo box at the bottom of the dialog box. Figure 8.7 shows the .EXE files. When you open a symbolics file the debugger then displays a Source Browser window, where you can set breakpoints and get prepared for the debugging session.

FIGURE 8.7

If you launch the debugger directly, you tell it where to find debug information.



Now that you know how to get all the necessary debug information created in the proper format and stored in the proper place, and you know how to launch the debugger, let's look at how the debugger itself works.

Understanding the Debugger Windows

The debugger is a powerful and full-featured application that does deep and exotic things with your code at a very low level in the machine, and as such it has a correspondingly complex interface. In this section we'll look at how the debugger presents information to you, so you can get to know the territory. Then we can talk about how to accomplish the common tasks you want to perform when debugging your code.

The good news is that much of the debugger interface will look familiar to you. You'll find a significant overlap in the design and functionality of the windows within the debugger, and between the debugger and the IDE (particularly the class browser). However, the overlap isn't 100 percent. The debugger does some things differently. The CodeWarrior tool set is constantly evolving, with major releases from Metrowerks every four months. Over time, more of the IDE interface will migrate into the debugger so that the differences will be minimized.

Like the IDE, the debugger has preferences. We'll discuss those as we encounter a feature or function that is controlled by a particular option. The debugger preference interface is a little different from the IDE, but you won't have any trouble figuring it out. The debugger also sports a full set of menus. Again, we'll discuss menu commands as we explore the debugger's many facets, because our focus is on how to do things, not what the menu commands are. The menu commands and preferences are only important insofar as they help you reach a goal. So, after talking about what you'll find in some of the debugger windows, we'll take a task-based look at what you want to do, and how to do it with the CodeWarrior debugger.

The debugger has two central windows, the Source Browser window and the Program window. We'll talk about each of those windows here, because they are the primary views you use when

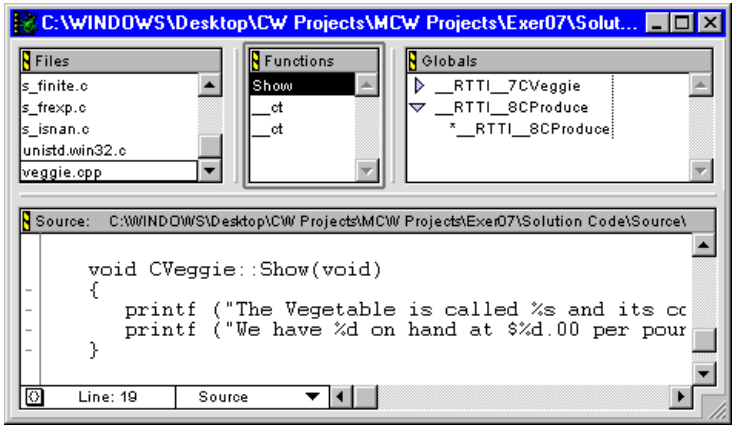
debugging code. In addition, the debugger presents a plethora of data display windows that serve a wide variety of functions. For example, you'll see data windows, a breakpoint window, the expression window, the process window, register windows, and more. We'll discuss these windows as we encounter them when we talk about how to do things with the debugger.

Understanding the Source Browser Window

The Source Browser window is the first window that shows up when you launch the debugger. If you launch the debugger from the IDE (with the Debug command in the IDE's Project menu), the debugger also opens the Program window. We'll talk about the Program window in just a bit. For now, let's stick to the Source Browser window shown in Figure 8.8. This is the window that appears when you open a symbolics file. The path to the file that contains the debug information appears at the top of the source pane.

FIGURE 8.8

The Source Browser window lets you work with any source code in the project.



Already you should feel pretty comfortable with the debugger, because this window looks a lot like the multi-class browser window

from Chapter 6. Much of its functionality is analogous to the browser as well, and that's not a surprise considering that this window and the class browser serve very similar purposes. The Source Browser window is really an interface that lets you browse through the symbolics information. The top left pane shows you every source file in the project, while the top center pane lists every function in the selected file, and the top right pane shows global variables.

You select the file you want to look in, and select the function whose code you want to see, using the file and function panes. You can use any of the standard techniques CodeWarrior uses for selecting items in a scrolling list. You can use keyboard navigation, point and click, or type ahead. When you select a file, the code in that file appears in the Source pane at the bottom of the window. When you select a function, the text on display in the Source pane scrolls to the selected function automatically.

NOTE

The symbolics file contains information on the path to your own source code and can open and display your own functions. If the file you select is in support code, such as in a code library, the debugger doesn't know where the source file is. In such a case you'll see the standard file dialog box asking you to locate the source file. You can navigate to the file or cancel the operation. If you cancel, the debugger displays assembly language code.

TIP

If you are debugging object-oriented code, the names in the Functions pane are listed alphabetically by class. You can list them alphabetically by function if you prefer. You can locate the option in the Debugger's Settings & Display preference panel. Toggle the Sort Functions by Method Name in Browser item in that panel. Figure 8.24 later in this chapter shows this panel.

The panes in the debugger are resizable, as they are in any CodeWarrior multipane window. Also, some familiar controls line up along the bottom of the Source pane. At the left end, the little icon with braces

on it displays a routine pop-up menu, analogous to the routine pop-up in an editor window. Although the button looks different, the utility of the routine pop-up menu is identical. Next to it is what looks like the familiar Line button that enables you to jump to any line in the file. In the debugger, this is not a button but a static display that shows you what line you're on. To the right of the line display is another little panel that says "Source." This is the pop-up menu that lets you choose between viewing the source code and viewing the assembly language object code.

So far, so good. Everything about the Source Browser's interface looks and acts more or less the same, so you know how it all works. The options have some subtleties, such as the line display not being a button, but overall, these differences are pretty minor. The big difference between this window and most of the similar IDE windows is in the Source pane.

Although the Source pane looks just like any other source pane you have seen in the IDE, it doesn't behave at all like the IDE. Two significant features distinguish the debugger's version of the Source pane from the IDE: one feature is missing, and one feature has been added.

First, the missing feature—the code in the Source pane is *not* editable. You cannot change code in the debugger, period. You can select code and copy it for pasting into other locations, but you cannot modify the code in any way. The source pane in the debugger is a static display—to change code you must return to the IDE and edit your code there.

Now, the new feature—the column to the left of the code. Next to each line of code you'll see a dash or line. This dash is what you use to set breakpoints. We'll talk about breakpoints in detail later in this chapter, but the ability to set a breakpoint is fundamental to your use of the debugger. So the means for working with breakpoints is right up front in one of the two principal debugging windows. (It's in the other one as well, as you'll see in just a moment.)

Now that you've seen what the window looks like, you might be wondering what you use it for. Well, the Source Browser window lets

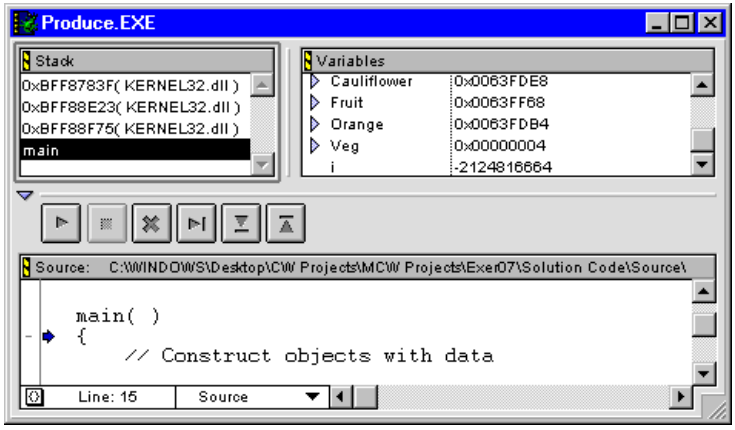
you navigate anywhere in your source code to view the underlying code and set breakpoints. We'll encounter this window again when we talk about how you accomplish both of those tasks—navigating code and managing breakpoints.

Understanding the Program Window

The second principal debugging view is the Program window. The browser lets you explore your code, but it has nothing to do with actually controlling your code as it runs. That's the purpose of the Program window. The Program window appears when you launch your program under debugger control.

If you use the Debug command from within the IDE, the Program window is where you end up automatically. If you launch the debugger directly from the Windows desktop and open a symbolics file, you must issue a Run command from the debugger's Control menu to see the Program window. Figure 8.9 shows you what this window looks like.

FIGURE 8.9
The debugger's Program window lets you control precisely how your code executes.



As you can see, once again you're looking at a browser-like window. Three panes appear in this window. The top left pane shows you the

stack or calling chain. The top right pane shows you the local variables for the currently selected function in the calling chain, and the bottom pane is the associated source code.

The Fundamentals of the Calling Chain

What is a *calling chain*? It is the series of routines that you passed through to get to where you are in your code.

As your code begins to execute, it is sure to make a call to a routine at some point. In turn, that routine may call another routine, and so on. Before long you have a whole series of nested routine calls that lead you deeper and deeper into your code. Each time a call is made, information about that call is placed in an area of memory known as the *stack*. In effect, each call's information is stacked below the previous call. It's rather like building a stalactite from the ceiling of a cave, only the length of the stack changes as calls are made and functions return.

This series of routine calls is the *calling chain*, and each time your code calls a subroutine, you move one step further down the calling chain. Each time you return from a subroutine, you remove the bottom link of the chain. In effect, the calling chain is the history of all the function calls that got you to where you are in the code. The currently executing routine is at the bottom of the list, and the routine that started your code is at the top.

In between the Source pane and the other panes you'll find a row of buttons. These buttons enable you to run, stop, or kill the application under debugger control, and to walk through your code line by line, stepping over, into, and out of functions at will. We'll talk about these controls in detail when we discuss how you control program execution using the debugger. A disclosure triangle lets you hide these buttons if you wish, giving you more room for source code.

TIP

Many programmers like to hide these buttons and gain the room because there is a corresponding menu command with a keyboard accelerator for every one of these buttons. You might prefer to master the keyboard controls and not use up screen real estate.

The Program window is just like the Source Browser window, except for the fact that it is missing the Files pane, and that it has the execution control buttons. You can resize individual panes and note that the Source pane in the Program window is noneditable. The purpose of the Source pane is to let you see your code so you can understand where you are and what's going on, not to edit your source. For that you can return to the IDE.

And most importantly, just like the Source Browser, a breakpoint column appears to the left of the code. In fact, if you set a breakpoint in one window it shows up in the other window as well. Both windows refer to the same list of information about where breakpoints are located.

Finally, the Program window includes one last unique feature—the current statement arrow. This little arrow, blue on color monitors, points to the line of code about to execute. Remember, the Program window is what you use to control program execution, so the current statement arrow always lets you know exactly where you are in your code. We'll talk about the current statement arrow again when we talk about how you control code execution.

These two windows—the Source Browser and Program windows—only begin to touch the surface of what the debugger can do. We'll meet the rest of the windows as we talk about how to work with the debugger.

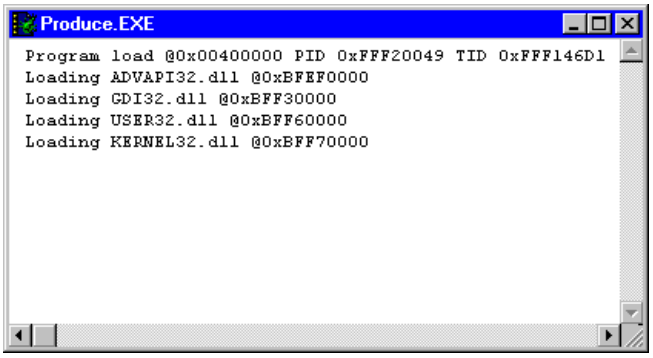
Understanding the Log Window

The log window appears whenever an event happens that the debugger thinks you might want to know about. It simply lists these

events, but this is an editable window so you can modify the contents, copy and paste, and save the contents of the window if you wish. Figure 8.10 shows a typical log window.

FIGURE 8.10

The log window keeps you apprised of what's going on with the debugger.



Mastering Basic Debugging

Before you can run, you must walk. Before you can learn about the truly cool and advanced features of the CodeWarrior debugger, you must learn about the basic functions it performs. Besides, ninety percent of the time what you'll be doing are basic things. You may never need the advanced features of the debugger such as multilanguage debugging. But you are certain to walk through your code, and step into and out of routines.

In this section, these are the kinds of tasks we concentrate on—the fundamental, ordinary, day-in-day-out things you do with a debugger. While they might seem mundane, this is where the real work gets done. Mastery of these skills really pays off on a regular basis. Best of all, because of the program's intelligent interface design, doing ordinary tasks with the CodeWarrior debugger is extraordinarily easy.

Stepping through Code

The first and most critical task of a debugger is to let you walk through your code line by line, so you can study what's going on as the program runs. Fundamentally, this is what the debugger is all about. Everything else it does depends on this prime directive. You may want to see variables, analyze expressions, set breakpoints, and much more, but unless you can control how the program actually executes, you aren't going to get very far.

Once you have the debugger launched and the program running (that is, you're looking at the Program window), you can do six tasks as you walk through your code:

1. Run the program
2. Stop the program
3. Kill the program
4. Step over a routine
5. Step into a routine
6. Step out of a routine

Table 8.1 lists each of the tasks, the associated menu item in the Control menu, and the keyboard accelerator. Figure 8.11 shows you the corresponding buttons in the Program window that issue the same commands. The disclosure triangle shows or hides these buttons.

In addition, the debugger implements a surprise treat—you can skip code altogether, or repeat segments of code at will. Let's look at each of these tasks, how to perform them, and any of the subtleties that might arise with respect to each of them.

FIGURE 8.11

The buttons in the Program window let you control all aspects of code execution.

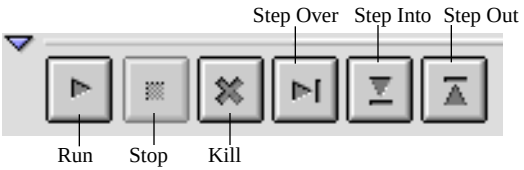


TABLE 8.1 The debugger code execution controls

Task	Menu Item	Accelerator	Function key
Run the program	Run	Ctrl+R	F5
Stop the program	Stop	Ctrl+period	Shift+F5
Kill the program	Kill	Ctrl+K	n/a
Step over a routine	Step Over	Ctrl+S (step)	F10
Step into a routine	Step Into	Ctrl+T (trace)	F11
Step out of a routine	Step Out	Ctrl+U (up)	Shift+F11

TIP

The function keys for the flow control commands are identical to Visual C++ 4.0.

Running Your Code

To get your program started, issue a Run command by whatever means you prefer. If you are in the Source Browser window, this command launches your application and displays the Program window, and stops you at the first line of code. Any initialization and setup code has already been executed. When you are in the Program window, a Run command actually starts your code executing.

While this seems simple enough, you have several additional considerations to keep in mind.

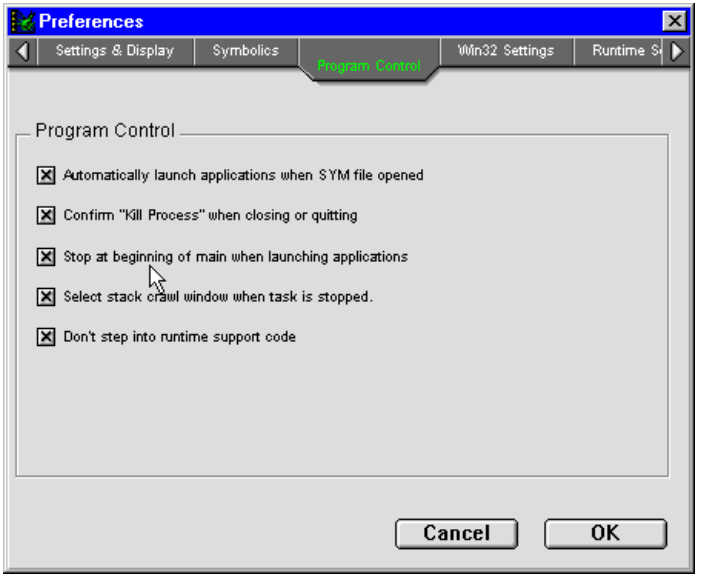
If you are launching an application from within the IDE, the process you just read about happens automatically. That is, you do not have to issue a Run command from the Source Browser window. CodeWarrior does that for you, and you see the Program window stopped at the first line of code. Similarly, if you open a symbolics file for an executable program (an application), by default the debugger auto-

matically runs the program and displays the Program window stopped at the beginning of the code.

Sometimes you want to stop and explore code that executes before the **main()** function is reached. For example, static constructors in C++ code are called before the debugger stops at the beginning of main(). To debug this code, you must turn off automatic running. In the Program Control panel of the debugger preferences, you can turn off the Automatically Launch Applications When SYM File Opened option. This option works for any symbolics file, of course, not just SYM files. Figure 8.12 shows you what this panel looks like. Or, on a one-time basis, you can hold down the Alt key while opening the symbolics file.

FIGURE 8.12

Use the Program Control preferences for the debugger to control whether the debugger stops at the beginning of the **main()** function.



Once the Source Browser window opens, you can set a breakpoint at the location where you want to stop, and then run the code. We'll cover breakpoints in just a bit.

When you do run your program under debugger control, you may not want your program to stop at the beginning of the **main()** function. You control this setting in the Program Control panel as well. Use the Stop at Beginning of Main When Launching Applications option to control whether the debugger stops at this location or not.

While we're talking about this particular panel, let me point out that there are additional settings you can modify with respect to program control. For example, you can make the stack crawl pane (the calling chain) the active pane when the program stops.

If you are debugging a library, you cannot run the library per se. You should use the Source Browser window to set a breakpoint at some spot conducive to your work, and then run some other executable that uses the library. When the code in your library gets called, the debugger interrupts program execution and stops at the breakpoint.

Finally, the Run command is useful for more than just starting your code the first time. Whenever your program execution has been halted for any reason, use the Run command to resume execution.

Stopping the Program

Getting things started is half the battle. Sometimes you'll want to stop your program so you can take a look at what's happening in your code. That's the only way you can find bugs! Fortunately, a Stop command is handy to tell the debugger to halt execution.

While this seems simple enough, once again some subtleties are worth keeping in mind. Here's the bad news—the Stop command is very imprecise. Program execution proceeds at an amazing speed, even under debugger control. When you issue a Stop command, you have no way to tell where in your code the debugger will actually stop execution. It could be just about anywhere—someplace interesting, or lost in the outer reaches of your code where you have no desire to be. It is almost certain that wherever you stop, you'll have to do some serious navigation to get to where you really want to be.

A better way to halt execution is to use breakpoints, a topic that I once again promise you'll learn all about in just a bit.

TIP

One nasty little bug that might show up when you run your code is an infinite loop. If you find yourself caught in one, you'll never reach the next breakpoint. The Stop command is very useful here. You should be able to switch back to the debugger and press the Stop button. You'll stop inside the infinite loop.

Killing the Program

If you *really* want to stop a program, the Kill command terminates a program completely—with extreme prejudice. This does more than stop execution, it closes down the program entirely. The Program window closes automatically because you have closed the running application. You can, of course, run it again from the Source Browser window if you wish, but when you kill a program that particular debugging session is over.

Killing a program is a rather drastic measure, and one you should normally avoid. For example, your code may need to do some work to clean up after itself as it shuts down. If you kill a program, it doesn't exit normally, it dies instantly and the cleanup work never happens. It is generally wiser to let the code run its course, or exit the program in normal fashion.

Because killing a running program is a rather severe thing to do, by default CodeWarrior asks you to confirm the request. You'll see a warning dialog box.

You can turn off this warning dialog in the debugger preferences, in the Program Control settings as shown in Figure 8.12. Toggle the Confirm "Kill Process" When Closing or Quitting option to turn this warning dialog box on or off.

Stepping over a Routine

In addition to the "all or nothing" commands to run, stop, or kill the code you are debugging, you can also step through your code one line at a time. This is where the debugger starts to become truly useful. As you step through your code, all the data displays available in the debugger are updated automatically. This lets you see exactly what's happening in your code each step of the way.

Stepping one line at a time is certainly a straightforward concept. The debugger command to do this is Step Over (Ctrl+S). When you issue a Step Over command, the current line of code executes and you stop, ready to execute the next line of code. When you issue the Step Over command, you step through one line of code.

So, why is this command called "Step Over"? It has to do with calls to subroutines. The line of code you execute may call another routine. If you Step Over, all the code inside that subroutine executes, and is treated as if it were one line of code. The debugger does not follow program execution down into that subroutine. The subroutine code executes, but the debugger doesn't stop program flow until the subroutine returns and the original line of code where you started completes execution. Effectively, you have "stepped over" the routine. It's like taking express, nonstop transportation from one city to another. You start at one point, end at another (the next line of code), and skip over all the countryside in between. You cross all that territory, but you don't stop there.

Remember, even when you step over a routine, the underlying routine actually executes. The Step Over command gives you a way to control exactly what you're seeing. If you aren't interested in what goes on inside the subroutine, you can skip it in the debugger.

Stepping into a Routine

On the other hand, you may be *very* interested in what goes on inside the called routine. In such cases, you want to follow execution into the subroutine. To do this, use the Step Into command.

The accelerator for this command is Ctrl+T. As a mnemonic for this, think of "tracing" program execution. When you issue the Step Into command, the debugger jumps to the first line of code in the called routine and stops you there. This is like taking local transportation from one city to another. You stop at every line of code along the way.

Stepping Out of a Routine

Finally, once you are in a routine, you may have seen everything you want to see and want to go back to where you came from to continue line-by-line execution. The Step Out command tells the debugger to run the rest of the routine, return to the calling routine, and stop. The accelerator for the Step Out command is Ctrl+U. As a mnemonic, think of taking a step "Up" the calling chain. That's exactly what you're doing. You're taking the express back to the last place you came from, and automatically running through any code remaining in the routine where you are.

That covers the six basic options you have for controlling program execution. However, you may recall that I mentioned a surprise treat in how you handle program flow—the ability to skip code altogether so that it doesn't execute at all, or to repeat the same code over again.

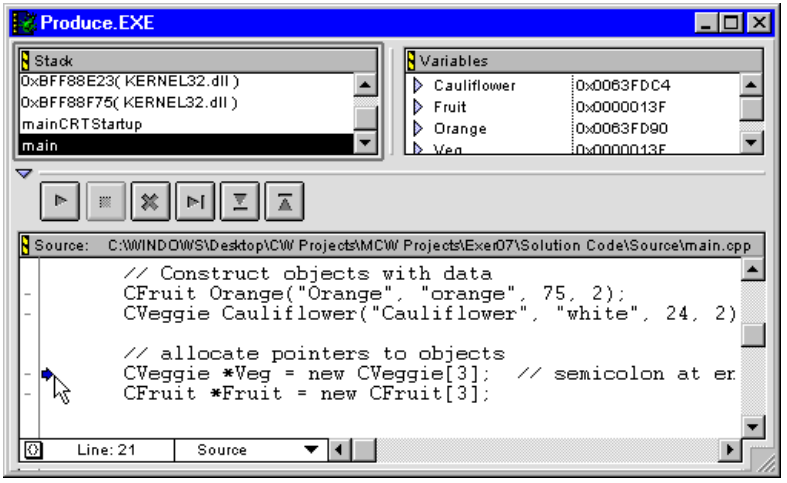
Skipping and Repeating Code

As you know, the Program window has a current statement arrow that indicates which line of code is about to execute next. This arrow is more than an indicator. It is in fact an active control that you can use to determine which line of code executes next. You can use the mouse to drag this arrow to any line of code in the file. As the mouse passes another line of code, the arrow jumps to that line. Wherever you position the arrow, that's the next line of code that will execute, as shown in Figure 8.13.

In effect, the arrow is equivalent to the program counter—the microprocessor's pointer to the next instruction to execute. Dragging the arrow changes the program counter. This is a very powerful feature,

FIGURE 8.13

The current statement arrow is relocatable, and points to the next line of code to execute.



and as with all things powerful, it is also dangerous. You have the freedom to drag the arrow to an entirely different function! You can imagine the havoc this creates with program flow.

More realistically, you would typically move the arrow to a different spot in the same routine. Even then you are at risk. The code in any given routine is frequently order-dependent. That is, for a line of code to execute properly, other bits of code that precede it in the program flow must run first. These earlier lines of code might initialize a variable, or perform some other setup task on which the particular line of code in question relies. If you drag the current statement arrow to a new line of code and skip critical setup work, the new line of code may not work properly. You can even crash the computer this way—and crashing is a Bad Thing.

However, the ability to drag the arrow to a new location is extremely useful. For example, if you suspect a bug exists inside a subroutine, you can skip the call (assuming it doesn't do anything critical) and see if everything still works. This can help you narrow your search for the bug. If you have a problem with a section of code, but you can't quite nail it down on the first pass through, you can simply drag the arrow back a few steps and repeat the code again, right there, immediately.

You don't have to exit the routine and reenter it from normal program flow. As long as you keep in mind the hazards to local variables and the necessity for every line of code to be set up properly, the ability to control the program counter in this way is a great tool.

Taken together, the three stepping commands—Over, Into, and Out—and the control you have over the current statement arrow are sufficient for walking anywhere in your code, but sometimes that's not good enough. And that brings us to the next topic, how to cruise through your code with speed and precision.

Navigating through Code

The task you perform most often in the debugger is traveling from one place to another in your code. While studying variables and other data is what you need to do to find and fix bugs, getting to the right spot to begin with is always a necessity. Realizing this, the debugger engineers created numerous techniques you can use to hop around, in tiny increments or in giant leaps.

Walking through Code

The simplest technique for navigating through your code is to walk through it line by line. You use the stepping controls discussed previously to accomplish this task. You can step through a line at a time, and you can move over, into, or out of any function with a single click or keystroke.

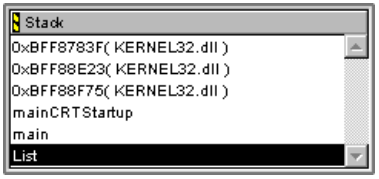
Stepping through code in this way is an excellent strategy if you want to move just a few lines, or go to a place very near your current location in your code. However, if you want to leap a greater distance, more efficient methods are available.

Walking through the Calling Chain

To see the code for any function in the calling chain, simply select that function in the calling chain pane. When you do, the precise spot

where you called out of that function down into the calling chain appears in the Program window's Source pane. All the local variables for that function appear with their current values. Figure 8.14 shows the calling chain pane of the Program window.

FIGURE 8.14
Use the stack crawl pane to instantly move to any function in the calling chain.



NOTE

The current value of a variable in a function higher in the chain may not be the value it had at the time of the call. The routines that have been called since then (lower in the chain) may have modified the contents of this function's variables. What you see is always the current state of the variables, which is as it should be.

The calling chain gives you a chance to quickly and easily see past history—to find out how you got to where you are. Seeing this history can be very helpful in tracking down program flow problems, as well as in finding other kinds of bugs. The principle is simple: If you don't know how you got to where you are, you'll have a hard time figuring out what's going on.

Besides, very often the point where a program actually collapses and dies is frequently *not* the place where the bug is. The bug can be, quite literally, anywhere in the code, and the damage it causes is a ticking bomb that goes off much later. While it can be anywhere, frequently the bug is higher up in the calling chain, so the ability to get back up the chain is extremely useful.

Browsing Anywhere in Code

The calling chain is a very common and likely place you'll want to look for worthwhile destinations. However, you will also want to go

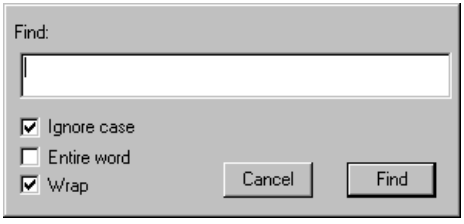
elsewhere in your code, and you can use the Source Browser window for this task. Activate the Source Browser window, and use the controls in that window to go anywhere in your code. You select a file from the Files pane, and that code appears in the Source pane. You can select a function from the Functions pane or the Function pop-up menu, and the Source pane scrolls to that function. You can use this window to go anywhere in your code.

However, the real point of using the Source Browser is usually to set a breakpoint in some file other than the one currently on display in the Program window. While you can see global variables when you switch to the Source Browser, you can't see any local variables in some other subroutine, because that routine isn't running. Although you can't see the code in action, you can use the Source Browser window just to study the code at some other place in your project, and that can be very helpful.

Finding Specific Code

The debugger also has a search mechanism to locate a particular bit of target text. Choose the Find command under the Edit menu, and the debugger's Find dialog box appears, as shown in Figure 8.15.

FIGURE 8.15
Use the debugger's Find dialog box to locate a piece of text in the active window.



The debugger's ability to find text is more limited than the IDE. You can enter a string of text, and the debugger looks in the file on display in the active Source pane (in either the Source Browser or the Program window). As with the IDE, you can make the search case-sensitive,

make it look for whole words, and wrap back to the beginning of the file for a complete search. The debugger's Edit menu also contains commands to find the current selection, and find the next hit. The debugger cannot search in multiple files, perform a batch search, or replace the found text. The debugger's find mechanism is a basic navigation tool you can use to get to a particular place.

Moving from Source to Debugger and Back

Finally, you'll experience situations when working in the IDE is best. After all, you're used to working in the IDE, so you know exactly how to find what you're looking for with the IDE tools such as the browser and its powerful search engine.

The best and most reliable way to work in the IDE is to switch back to the IDE as a separate application. This gets you back to the copy of CodeWarrior that has your project open. You can then use the IDE to locate and edit the file.

If you want to open one file to make a quick change, however, you can do that directly from the debugger using a couple of techniques. Double-click a file name in the Source Browser window's File Pane and the file opens in Notepad. Or you can choose the Edit *File name* item from the File menu (*File name* will be the name of the file on display in the active Source pane). However, you're missing all the code-related features of the IDE's source code editor.

In sum, the debugger gives you the ability to walk through code line-by-line, browse and jump to anywhere in your code, or use the IDE as a code navigator to tell the debugger what code to display. These powerful features combine to give you all the navigational power you need to get to where you want to go in your code quickly and accurately.

TIP

Actually, you can specify any editor application in the Win32 Settings panel in the debugger preferences. The debugger will launch a copy of the specified application, which may or may not be able to open and display the file. Even if you specify CodeWarrior, the debugger opens a new copy of CodeWarrior, not the one that already has your project open. Manually switching back to the running copy of the IDE is still the best way to edit your files while working in the debugger.

Working with Breakpoints

Of course, the principal reason why you want to go somewhere in your code is so you can stop execution and take a leisurely look around, see what's going on, study your variables, and hunt for bugs. After all, it isn't called a debugger for nothing.

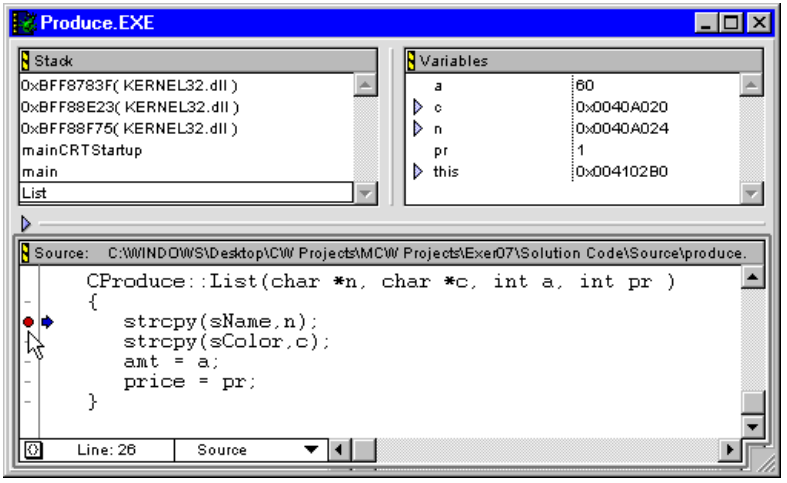
As a practical matter, the Stop command is useless for this purpose. The real way to stop execution at a planned spot is to set up a breakpoint. In this section you'll learn how to set and clear a breakpoint, attach a condition to it, use a temporary breakpoint, and more.

Setting and Clearing a Breakpoint

As you have already seen, both the Source Browser and Program windows display a breakpoint column to the left of the source code that appears in the Source pane. A dash appears next to each line of code. To set a breakpoint, you simply click the dash, and a red circle appears indicating that program execution will stop when it reaches this line of code. Figure 8.16 shows a breakpoint set in code.

You clear a breakpoint the same way. Simply click the red circle, and it becomes a dash again. You can clear all breakpoints with the Clear All Breakpoints item in the debugger's Control menu. Adding and removing breakpoints is that easy.

FIGURE 8.16
Click the breakpoint column next to a line of code to set or clear a breakpoint.



The Fundamentals of Breakpoints

A *breakpoint* is a stop sign or flag that tells the debugger (which watches every line of code as it executes) to stop when it reaches a particular spot. This is vital to your work, because you can tell the debugger "I want to go up to here and no further." You can race over everything that comes before without stopping, but once you stop you can take a good look around.

In addition, you'll want to use temporary and conditional breakpoints. A *temporary* breakpoint stops you one time, and then disappears. We'll have more discussion of temporary breakpoints later. A *conditional* breakpoint says to the debugger, "Stop here, but only if a certain condition is met." This lets you narrow the focus of the breakpoint so it doesn't kick in every time the debugger hits that line of code.

You can set a breakpoint in either the Source Browser or Program windows, it doesn't make any difference which window you use. This means you can use any debugger navigation technique in either window to get to a particular place in the code to set a breakpoint.

Using a Temporary Breakpoint

Sometimes you are looking at a place in code and you just want the program to stop there once the next time it arrives at that spot. If you Alt+click the breakpoint column next to a line of code (once again, in either the Source Browser or Program window), you set a temporary breakpoint. A red circle appears, so a temporary breakpoint looks just like a regular breakpoint—you can't tell them apart by looking at them. However, a couple of other things happen when you Alt+click to set a temporary breakpoint.

Most surprisingly, program execution resumes automatically and immediately, without issuing a Run command. Execution continues until that line of code is reached. When the debugger reaches the temporary breakpoint, execution stops and the debugger clears the breakpoint, again without any further action on your part.

In the end, you have accomplished your goal with a single click. You got the program to run and stop at exactly the point you want. And no breakpoint remains behind to clutter up future program execution.

NOTE

If the debugger encounters another breakpoint along the way, of course it stops there first, but the temporary breakpoint continues to exist until it is reached.

The Mystery of the Missing Breakpoints

Sometimes when you look at code in the debugger, you'll notice that no dash appears next to a line of code. What gives? You're looking at your source code, and you certainly remember writing it, yet you cannot set a breakpoint in the object code! It's as though your object code has mysteriously disappeared. Where did it go?

Object code can disappear like this for two reasons—dead-stripping code, and optimizing code.

When you link a project, the linker checks to see whether each routine in your code is ever used. If a particular routine is never called from anywhere, and could never be entered, the linker strips that code out of the final executable—and the object code disappears. If that happens, when you open up a source file in the debugger you can still see the lines of source code you wrote, but no dash will appear where you can set a breakpoint, because object code no longer exists that corresponds to the source code. Of course, you're not really missing anything important. Even if the object code were there and you set a breakpoint in it, that breakpoint would never be reached because the code would never be called.

The second reason why you might not be able to set a breakpoint at a line of code has to do with the optimizing compilers. Just about everything in the debugger, and especially breakpoints, depends on a one-to-one relationship between your source code and the final object code in your program, as preserved in the symbolics file. As the debugger walks through the object code, it locates and displays the corresponding source code in the various debugging views.

The CodeWarrior compilers are optimizing compilers. You can turn optimizations on or off at various levels and in various ways. If you optimize your code, the compiler starts playing around with details in your

continued ➠

The Mystery of the Missing Breakpoints

code to make it faster, or smaller, or more efficient. For example, your source code might set a variable to a value inside a loop, but that value never changes. (This particular problem is known as a loop invariant.) Your code will run faster if that value is set once *before* the loop executes rather than being set every time through the loop. So, to give you the speed gain the optimizer moves the offending instruction out of the loop.

The compilers use all kinds of optimizing tricks to enhance the speed of your code. However, when a compiler applies these tricks, your code can move around. Variables can disappear, new variables you never thought of might be created to serve some purpose, two or three variables may end up sharing the same memory space, individual instructions may be reordered to take advantage of instruction-scheduling features of the processor, and so on. In the end, the object code may bear very little resemblance to the original source.

Now, think about what this does to the one-to-one relationship between object code and source code. It is destroyed—for good cause, yes, but destroyed nonetheless. If you've optimized your code, quite likely you will be unable to debug it with the source-level debugger. The debugger will become terminally confused about which object code belongs to which line of source code. In fact, in many cases breakpoint dashes won't even be available for lines of optimized code. You won't be able to walk through your code, or stop where you want, because the object code has been thoroughly scrambled.

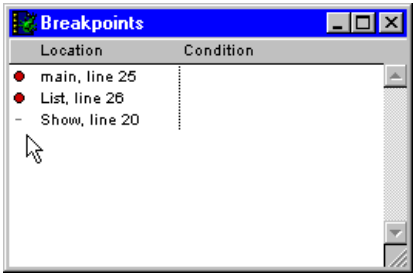
The solution to this problem is simple—don't optimize code you are going to debug. Turn off all optimizations until your code is thoroughly debugged, and then you can optimize it and test it in its final form.

Viewing and Controlling Breakpoints

As you work with code in a project, you can amass quite a collection of breakpoints. You can see every breakpoint you have set in your

code, gathered together and displayed for you in a single window called the Breakpoints window. The debugger's Window menu contains the Show Breakpoints command. Figure 8.17 illustrates what the window looks like.

FIGURE 8.17
The Breakpoints window shows all breakpoints in a project, and lets you turn them off without clearing them.



Each breakpoint has a line in this window listing the file that contains the breakpoint, and the line in that file where it is located. To see that breakpoint in the Source Browser window, just double-click the breakpoint. This navigation aid lets you jump instantly to any existing breakpoint. You can also clear breakpoints here if you wish. Simply select the breakpoint in the Breakpoints window, press the Backspace key, and the breakpoint is removed from the code.

When you have several breakpoints, they can start getting in your way. You may be working on two or three sections of buggy code simultaneously, and breakpoints you set to study one problem could interfere with program flow as you attempt to study another. You could clear the troublesome breakpoints, but you know you're going to want them back again a little later. The most time-consuming part of setting a breakpoint is locating the spot where you want to place it. First, you have to navigate to the general area, and after that you have to look at the code and figure out the best spot to stop. Sometimes it's obvious, but sometimes figuring out the right spot takes time. Having to do all this work more than once is inefficient, but when you clear a

breakpoint it's gone forever. Luckily, there's another way to deactivate a breakpoint without destroying it irrevocably.

In the Breakpoints window (and this might be its nicest feature), you can activate or deactivate a breakpoint *without* clearing it. If you click the breakpoint column in the Breakpoints window, you turn off the breakpoint, but the marker for that breakpoint continues to exist. This lets you shut off the breakpoint without actually removing it from the file. Later on, you can come back to the Breakpoints window, turn it back on, and you're back in business without having to find just the right spot all over again.

TIP

You can use breakpoints as handy markers in your code. Create them at important places where you expect to return regularly—like bookmarks. Then use the breakpoint window to jump to these spots by double-clicking the breakpoint entry in the list. If you don't want program flow to stop, just turn the breakpoint off—you can still use it as a marker.

Creating a Conditional Breakpoint

The Breakpoints window is also where you attach a condition to a breakpoint. What is a conditional breakpoint? It is a breakpoint that works only if some attached condition is true.

For example, let's say you have a loop in your code that repeats one hundred times, and you know that there's a problem that appears after ninety times through the loop. You need to set a breakpoint inside the loop so you can study what's going on. You certainly don't want to break ninety times just so you can get to number ninety-one. This is where a conditional breakpoint comes into play. In this example, you would use whatever variable is your loop counter, and set a condition that says break at this spot if the value of this counter is greater than ninety. Otherwise, keep on going.

Every time it encounters a breakpoint, the debugger looks to see if a condition is attached. If a condition is attached, it must be an expression of some kind that ultimately resolves into a true/false condition. If the result of analyzing the expression is true or nonzero, the debugger stops at the breakpoint. If the condition is false or zero, the breakpoint is ignored. For example, an expression such as **loopCounter > 90** would work just fine.

You can only have one condition attached to a breakpoint, but that condition can test for multiple situations. For example, you might have an expression that says **((gDone == false && top <= 10) || bottom > 20)**. This one expression tests a variety of variables to see if the overall condition is met. The only requirement is that, ultimately, the entire expression resolves to a numerical or true/false value.

To set a condition, double-click in the Condition column to the right of a breakpoint, and type in the condition. You can also use cut and paste techniques, or drag an expression from some other source into this column. If the expression uses variables (and most do), that variable must be in scope for the code at the breakpoint. In other words, it must be a variable that actually exists at that point in the code. To clear a condition, simply select it and delete it.

Breaking on Exceptions

You can stop program execution in yet another way, and that is to break on an exception. The C++ and Java languages have a formal error handling mechanism called *exceptions*. When an error occurs, your code can throw an exception. Of course, errors are frequently related to bugs, and finding bugs is what a debugger is all about. Stopping whenever an exception is thrown becomes a particularly useful thing to do.

Consequently, the CodeWarrior engineers put in a command to Break on C++ Exceptions, and to Break on Java Exceptions. You'll find each command in the debugger's Control menu. The commands are active when you are working in code using the corresponding language. If these features are activated, you don't need to set a break-

point to catch an exception. CodeWarrior stops automatically whenever an exception is thrown. You can then go exploring right from the exception. Now that's friendly.

This simple and direct control of breakpoints—setting, clearing, viewing, activating, deactivating, and jumping to the related code—makes working with breakpoints a snap in CodeWarrior.

Viewing and Changing Data

So far we've covered how to move around in your code using the debugger, and how to make sure you stop at the right spot reliably. Of course, once you get to where you want to go and you have stopped, you want to take a look around and see what's happening.

Viewing and modifying variables are both critical and common tasks you must perform in the debugger. The debugger tries to be as friendly as possible in this respect. It lets you see both global and local variables at any time, see them with their data types, and display the value in the variable in a variety of formats. You can even change the contents of a variable quickly and easily.

Seeing Variable Values

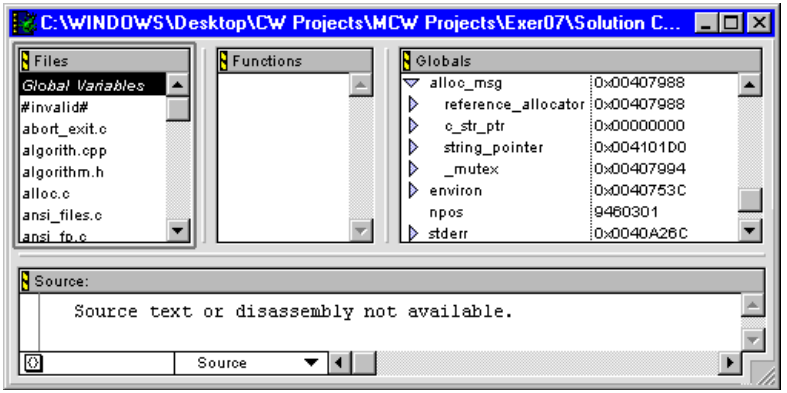
Just being able to see what the values in your variables are is one of the greatest diagnostic attributes of the debugger. Sometimes a quick glance at a variable's value is all you need to identify a bug. Both global and local variables are readily available in the debugger.

You can see global variables in the Source Browser window, in the Globals pane. To display globals, first look at the top of the file list and locate the entry for the global variables. When you select this item, all the globals appear. Figure 8.18 shows you how it works.

You can see local variables in the Program window, in the Variables pane as shown in Figure 8.19. In either the Globals pane or the Variables pane, the name of the variable appears on one side, and the val-

FIGURE 8.18

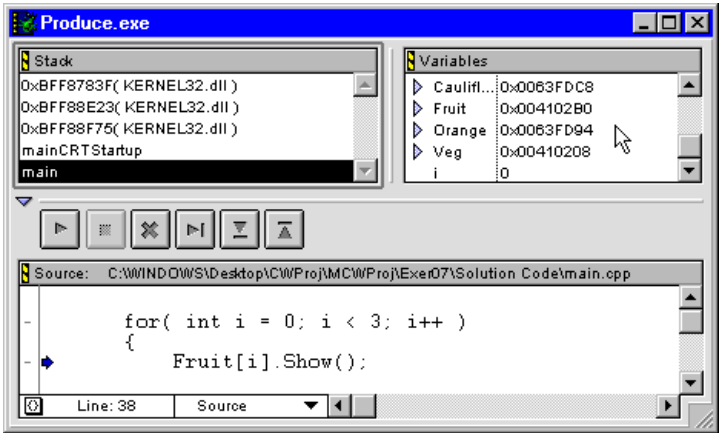
You can use the Source Browser window to display all of a project's global variables.



ue on the other. A thin vertical line separates them, and you can drag this line to resize the name or value columns within the pane.

FIGURE 8.19

Use the Program window to see all local variables.



Variables that are structures of some kind (such as a pointer to a class or struct) have disclosure triangles to the left of the names. Let's call these *hierarchical variables* for the sake of this discussion. You can show or hide the fields or members of such variables by clicking the trian-

gles. You can also select any variable, and use the arrow keys to move around in the list at will.

TIP

The disclosure triangles work as in the browser. Alt+click a disclosure triangle to show or hide all levels inside a hierarchical variable. Use the Collapse All item in the debugger's Data menu to hide all the contents of all hierarchical variables. Open all hierarchical variables at the same level with a Ctrl+click. Do a wide and deep disclosure with Ctrl+Alt+click.

When you leave one routine and enter a new routine while stepping through code, a new set of variables appears—those that are in scope for the new function. All hierarchical variables are always collapsed when you enter a function, even if you expanded some or all of them on a previous occasion when you were in the same function. The debugger does not remember the expanded state of hierarchical variables from one entry to another. This is really aggravating, but a wonderful solution to the problem waits around the corner—the Expression window. You'll see how to use that window in just a little while.

Setting Variable Values

In addition to seeing each variable's value, you can also change the value of any variable. Select the *value* (not the variable name) and press the Enter key, or double-click the value. Either way, the value becomes an editable text field and you can enter any value you want for a variable.

Being able to set a value for a variable at any time enables you to test your code with arbitrary contents, not just with the value in the variable as the result of ordinary program flow. You should be careful what you enter as a value, however, because the debugger cannot prevent you from entering a value that will crash the machine. If you are not careful, you might want to expect the unexpected.

When you enter a new value, you must enter it in a form the debugger understands. The debugger recognizes data in the following forms:

The Joy and Aggravation of Register Coloring

Compilers are weird and wonderful things, and compiler writers deserve a special place in programmer heaven for giving us such great tools (and saving the rest of us from having to master low-level machine code). Compiler writers like their compilers to create fast, tight, efficient code. That's why they invent so many optimizing tricks. *Register coloring* is an optimizing trick, but it's one that you cannot turn off in the Metrowerks compilers. Register coloring is on all the time.

So what is it? Here's the scenario. You create a routine, and it has a series of local variables. For the sake of this discussion, let's say the names of two variables are "red" and "blue." In your code, you use these two variables, but they never overlap. That is, you set a value inside red and use it. A little later on in this routine, you set a value inside blue and use it, but red is never heard from or used again after you start using blue.

So much for the source code. Now for some nuts-and-bolts details at the object code level. The compiler likes to store a local variable in a register—right inside the processor—because access to a register value is *much* faster than access to a value in RAM. The problem is, only so many registers are available.

Well, the compiler writer looks at your code, and at how you use red and blue and says, "Why should I use two variables? I can do the same thing with one." Where you created two variables in your code to keep things neat, clean, and easily understood, in fact one will do because your variables are never used simultaneously. So the compiler writer eliminates one of them, thus saving a register. This is register coloring—the same register is both red and blue, but at different times. When your source code works with the red variable, the object code sets a value in a particular register. Later on when your source code works with the blue variable, it uses that same register all over again, because it knows you're never going to use red again.

continued ➠

The Joy and Aggravation of Register Coloring

What does this do to the debugger's display of a local variable? Well, when you look in the Variables pane, you'll see both of your faithful variables red and blue, and each has a value. But if you watch closely, a strange thing happens. When your code sets the initial value for red, the value for blue changes as well. Although they have separate names, they are in fact both referring to the same location to get their value, so the two variables work in lockstep—they are really one variable with two names.

When you see this happen, your first response is to reach for the bug killer, because you're sure you've found a problem. As far as you're concerned, you have two variables, not one. And two separate variables aren't supposed to behave that way! Ah, but with register coloring, they do.

Don't go running back to your source code and eliminate what you think is a useless variable. Using separate variable names, even if they don't overlap in a single routine, can really help make your source code much easier to understand. Besides, the compiler fixed the problem for you! You haven't wasted space, time, or memory because the compiler cleaned up after you.

While register coloring can lead to a little weirdness in the debugger, just remember that the compiler writer has saved you from having to program in assembly language.

- **Decimal**—digits 0 through 9, with or without sign (exponent entry is fine, as in 32e4)
- **Hexadecimal**—0x followed by digits 0 through 9 and A through F, with or without sign, as in 0x7A3D
- **Floating-point**—digits 0 through 9 using a decimal
- **C string**—a string of text inside double quotes, as in "This is a C string" (the debugger adds the terminal zero)

- **Pascal string**—a string of text (255 characters maximum) starting with \p inside double quotes, as in "\pThis is a Pascal string" (the debugger sets the size of the string)
- **Character constant**—a character or characters inside single quotes, as in 'char'

After you enter a value, the debugger converts it to the format being used to display that variable's value. For example, if you are changing a variable displayed as a decimal value and you enter a hexadecimal digit, when you press Enter after typing in your new hexadecimal value the debugger displays the value in decimal form.

Of course, if you really want to see the value in hexadecimal form, the debugger can do that too.

Viewing and Changing Data Types

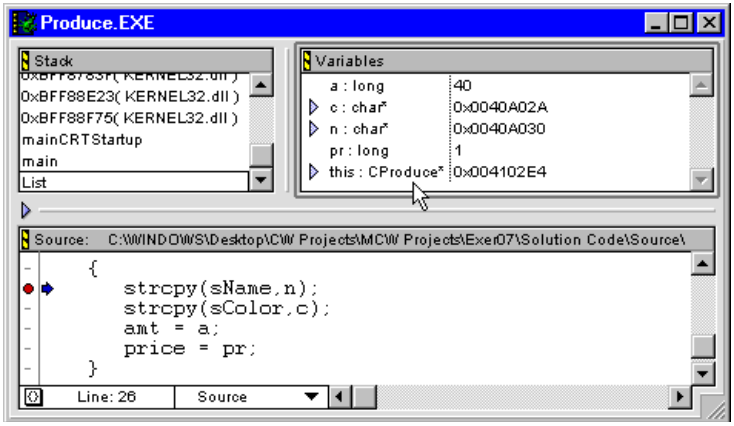
Every variable contains data of a particular type, such as an integer, a floating point number, a string, and so forth. The true nature of the variable is not always evident from the data it contains, and the variable's data type can be significant information. You can tell the debugger to display each variable's data type as well as its name by choosing the Show Types item in the Data menu. When you do, the data type for each variable appears as part of its name. Figure 8.20 shows this configuration.

You can make the display of data types the default standard for the debugger. The option is named In Variable Panes, Show Variable Types By Default. You can find this setting in the debugger's Settings & Display preferences panel as shown in Figure 8.24 a little further on in this chapter.

The debugger does its best to present a variable's value to you in the format that makes the most sense. For example, the debugger displays an address as a hexadecimal value, displays integer data types in decimal numbers, and displays strings as text. However, sometimes you want to see data as some other type than the one the debugger thinks is best. You might want to see the decimal equivalent of an address for

FIGURE 8.20

You can see each variable's data type in the debugger if you wish.



example, or an integer data type displayed as a hexadecimal value. To do this, simply select the variable or its value, and then choose a display format item from the Data menu. The choices available to you are:

- Default
- Signed Decimal
- Unsigned Decimal
- Hexadecimal
- Character
- C String
- Pascal String
- Floating Point
- Enumeration
- Fixed
- Fract

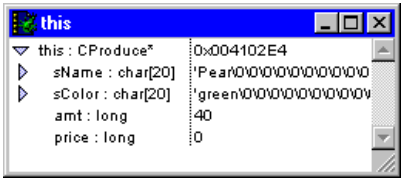
All choices are not available for all data types. The debugger lets you choose those types that it can use to display the variable you have selected. However, after a variable goes out of scope, the next time it comes back into scope the debugger uses the default display type. As

with the expanded state of hierarchical variables, the debugger forgets what you did last time, and this can be very annoying. Every time you return to a function, you have to repeat the setup work to see the contents of the variable the way you want. As with expanding or collapsing individual variables, the Expression window solves this problem too, and we'll talk about it in just a bit.

Displaying a Variable in Its Own Window

You aren't restricted to the fairly limited display area of the main debugger windows when you want to examine a variable. You can also open a separate window to display any single variable if you wish. Just double-click the variable name in the debugger (or select the name and choose the Open Variable Window item from the Data menu). When you do, the debugger opens a window and displays the variable in the window. Figure 8.21 shows such a window.

FIGURE 8.21
You can display any variable in its own window to make it easy to see the value.



Putting a variable in its own window is a great way to save time, especially for global variables. You don't have to repeatedly set the display type, and you can keep it on display across a whole series of functions, so you can watch how the value changes throughout program execution.

Displaying a local variable in its own window can be useful as well. You can isolate a particular variable so you can see it more easily. For example, you might have two variables you want to track, and you haven't got enough room in the Variables pane to see them both. You can put one or the other in it its own window—or both for that matter.

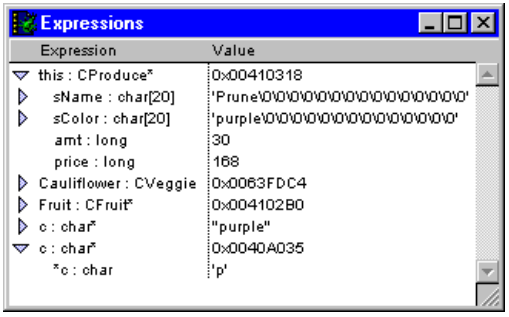
The downside to this is, if you want to look at a collection of variables, you can quickly become lost in a sea of little windows. And, when the variable goes out of scope, the window disappears automatically! That means the next time you return to a function, you still have to do this setup work all over again. Well, as with hierarchical variables and displaying data in nonstandard types, the solution is the Expression window.

Using the Expression Window

The Expression window is a single place where you can gather all kinds of data from all over the debugger, thus saving yourself a lot of setup work—and a lot of eyestrain as you hunt for something like a local variable. Figure 8.22 shows you what the Expression window looks like when you choose the Show Expressions item from the Window menu.

FIGURE 8.22

The Expression window is the debugger's one-stop variable-shopping extravaganza.



You can add *any* variable, local or global, to the Expression window. To do so, select the variable and choose Copy to Expression from the Data menu. You can also drag it from any other window into the Expression window. Once you have items in the window, you can reorganize them by dragging them around. To remove an item, simply select it and press the Delete key.

The Expression window works very much like the individual window you can open for any particular variable, only you get to collect a whole bunch of variables in one place. The name of each variable or expression appears on the left side of the window, and the value appears on the right. Of course you can edit values right here.

Best of all, unlike the individual variable window, when the function that defines the variable goes out of scope, the variable remains in the Expression window. The value is automatically updated when the debugger stops program execution. (Any variable currently out of scope is left blank.) Your display choices with respect to data display and expanding hierarchical variables also remain unaffected. If you expand a class or struct, it stays open, and if you set the display type for a value, that setting sticks. That's nice.

TIP

Here's a neat Expression window trick. If you have a variable that you'd like to see as two different data types (say decimal and hexadecimal), put two copies of the variable in the Expression window. Set one to display as one data type, the other as your second choice. You get to see them both simultaneously. Cool!

The Expression window can hold more than just variables. It can hold any expression, however complex. For example, if you have a multipart conditional test that uses Boolean logic to compare a series of variables, you can enter that expression into the Expression window and see how it is affected as the program runs. Clearly the Expression window is a sweet tool.

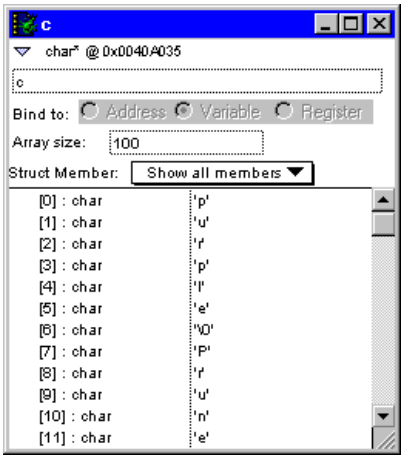
Working with Array Variables

Some variables are really arrays of values, rather than a single value. CodeWarrior uses a special window to give you easy access to any value in an array. To see this window, first you must select a variable that is in fact an array, and then choose the Open Array Window item in the Data menu. This window gives you access to every element in the

array. The array window, shown in Figure 8.23, hides some interesting twists.

FIGURE 8.23

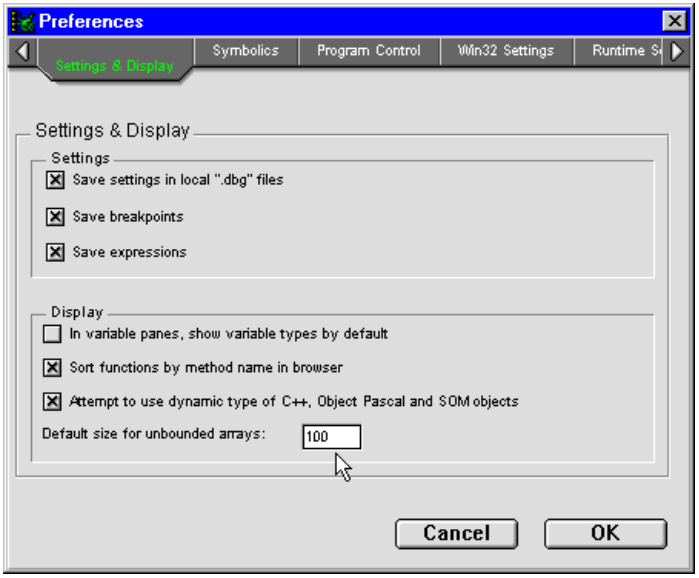
The Array window lists memory starting at an address, and interprets the contents of memory as array elements.



At the top of the window is the data type for this array, and the starting address of element zero of the array. Next to this is a disclosure triangle you can use to display the information pane for this array. This pane is visible in Figure 8.23. You can set the array size, and determine how the array window is "pinned" to memory. It can be pinned to a variable, to an absolute address, or even to a processor register. If it is pinned to a variable, the array window closes automatically when the variable goes out of scope.

In the C/C++ language, arrays are unbounded. The debugger has no idea how big the array is supposed to be, so it simply displays a series of values in memory, starting at an initial location. By default, it assumes the array has 100 elements. You can set the default number of elements in the debugger's preferences. In the Settings and Display panel, set the value in the Default Size for Unbounded Arrays. Figure 8.24 shows this option.

FIGURE 8.24
Set the assumed size for C/C++ arrays in the debugger's Settings and Display preferences panel.



Saving Your Debug Information

As you have seen, the debugger has many ways to view your data. Some of them, such as the Expression window, can require a lot of work to set up. You might have to collect variables from several locations. Similarly, putting all the breakpoints in place can be a chore if you have a lot of them.

The CodeWarrior debugger lets you save your debugging arrangement so you don't have to go through all the work of restoring it next time you want to debug the same project. You save all these settings with the Save Settings item in the File menu. The debugger saves the settings in a file with the same name as your project, but ending with the .dbg extension.

Precisely what gets saved is controlled by the settings section of the Settings & Display panel you just saw in Figure 8.24. You can save settings (window size and location), the breakpoints, and (best of all) expressions in the Expression window.

NOTE

If any of these three options is on—settings, breakpoints, or expressions—the debugger creates the .dbg file to store the information. The Save Settings in Local ".dbg" Files option just controls whether window size and location are saved in the file, not whether the .dbg file is created.

Although you can explicitly save settings at any time (and it might be wise to do so from time to time in case your code crashes and brings down the machine and the debugger with it), settings are also saved automatically when you quit the debugger. So, if you forget to explicitly save your settings, the debugger takes care of it for you.

If you save breakpoints, and then subsequently change your code, the information in the .dbg file will likely no longer correspond to the source code. If this happens, the debugger displays a warning when you try to run the code telling you that some breakpoints could not be restored. You can then clear them all, or just accept the warning.

The flexibility the debugger gives you is immense. You can go wherever you want, whenever you want, set breakpoints anywhere, see any and all variables that are in scope at any moment, and set their values at will. You can gather up tidbits from all over your code, put them in one place for easy viewing, and even tell the debugger how you want the data displayed.

But the debugger doesn't stop there. Because it can access your code and the internals of the machine at a low level, it can give you still more power to wander around in the electronic netherworld.

Working with Low-Level Debugger Features

While at its core the debugger is a high-level debugger designed to make your debugging experience as painless as possible, the CodeWarrior debugger does have several nice features commonly found only in low-level debuggers. A low-level debugger has direct access to memory, including processor registers, and usually works primarily (or only) in assembly-language instructions. CodeWarrior has the ability to do all three of these tricks—view memory, view processors, and work in assembly language.

Working with Raw Memory

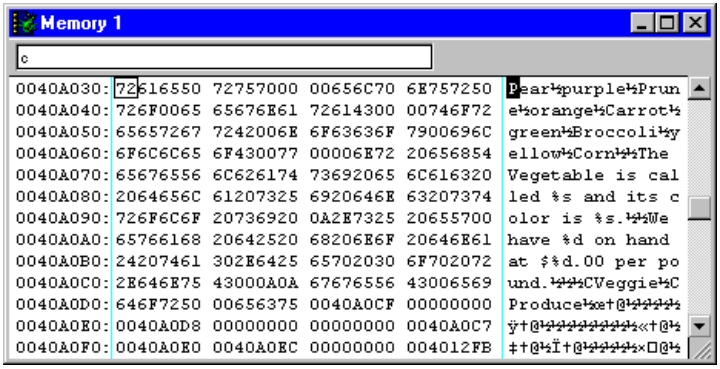
The debugger gives you complete access to memory, raw and unfiltered. Once you're in, you can do anything you want. This feature is amazingly powerful—and correspondingly dangerous.

To see the Memory window, choose the View Memory command from the Data menu. Figure 8.25 shows what the resulting window looks like. At the top of the window notice an editable field that holds an address, or the name of a variable. The value of this expression will be the beginning address of the memory view, and you can enter any expression you like. For example, if you select a variable that contains a memory address, such as a pointer data type, and then choose the View Memory command, that variable becomes the initial address. The view shows you both hexadecimal contents and corresponding ASCII values.

All data in the Memory window is fully editable. Simply use the mouse to set an insertion point, and start typing. If you click in the hexadecimal display, you must enter hexadecimal digits. If you click on the ASCII side of the display, you can enter ordinary text. You can't delete any values (after all, memory must hold something) but you can replace them with new values.

FIGURE 8.25

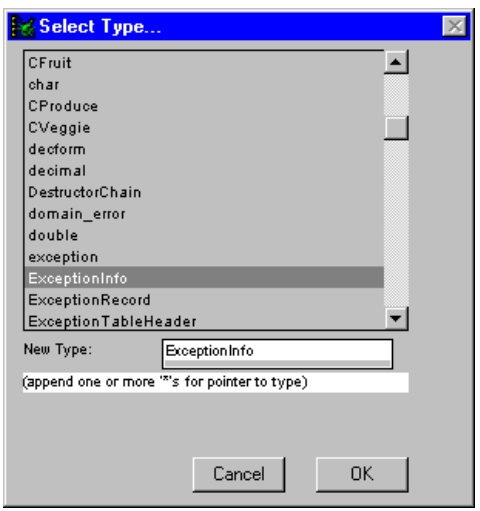
The Memory window gives you complete and unfettered access to any location in memory.



You can also use the View Memory As item in the Data menu to specify a data type that the debugger should use to interpret the values in memory. This command opens an Array window and displays memory as if it were a series of elements of the selected data type. You select the data type from a dialog box, as illustrated in Figure 8.26. The data appears in the same kind of Array window we just discussed above, and is fully editable.

FIGURE 8.26

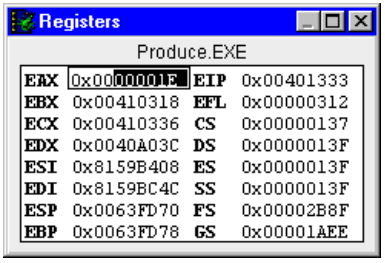
You can pick the data type used to display and interpret a range of memory, instead of looking at raw data.



Viewing and Changing Registers

If the preceding is not power enough, the debugger also gives you free access to each processor register. To see the registers, choose the Show Registers item from the Window menu. The window shown in Figure 8.27 appears, listing each register along with its current value. If you double-click a value (or select it and press Enter) you can change the value in that register.

FIGURE 8.27
The Registers window gives you unlimited access to every processor register.



The actual appearance of the Registers window depends upon the target processor, because each processor has a unique set of registers. Nevertheless, each register for your host is there and available. If your processor has a floating-point unit (FPU), you can see the FPU registers as well. The Show FPU Registers item in the Window menu is enabled when an FPU is present. Again, you can modify any register value.

WARNING

The processor registers are critical data and instruction storage areas that control the entire computer. Mess with some of these, and I guarantee your program will crash big time—so be careful out there.

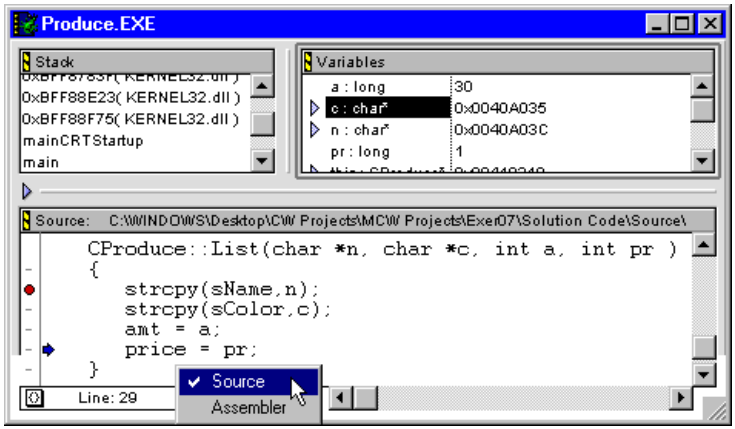
Viewing Assembly Language

Most source-level debuggers let you see your source code as you walk through your project. The CodeWarrior debugger lets you toggle between seeing the original, high-level code or the assembly-language

instructions for your target. Switching between the two views is simple. You just click the pop-up menu at the bottom of the Source pane in either the Program window or the Source Browser window. Figure 8.28 shows this simple menu. Just pick the viewing mode you want, source or assembler. This feature gives you immediate access to your code's disassembly at any time.

FIGURE 8.28

The code view pop-up menu lets you view your code either in a high-level language or assembly-language instructions.



The debugger is fully functional in assembler mode. You can navigate through your code, set breakpoints, see and modify variables, move the current statement arrow—the works. In fact, the assembly language view is the *only* view you get if the source code is unavailable for any reason. For example, you might have debugging information turned off for a particular file, and then attempt to step into a function in that file. You can do so, but you'll see assembly instructions, not source code.

Remember, the CodeWarrior debugger is designed to work on multiple hosts and handle code for multiple targets. Depending upon your target, you might see Java bytecodes, or the assembly instructions for Intel x86, PowerPC, Motorola 680x0, or MIPS. The debugger tool is clearly multilingual as well as multitalented.

Debugging Libraries

The debugger can handle dynamically linked libraries (DLLs) without a problem. To debug a DLL, you launch the debugger as a stand-alone application, and open up the symbolics file for the DLL. Once you have the Source Browser window open, you can set one or more breakpoints at places in the library code where you would like to stop and take a look around.

After this, launch some application that uses the library. When the debugger encounters a breakpoint in the library code, the debugger stops execution of the library function and gives you control. At that time you can debug the library code in exactly the same way as you would an application.

Cool Features

You might be wondering what the debugger can do for an encore. After all, with such features as unlimited access to processor registers and the Expression window to gather up data from all over your code for easy viewing, how much cooler can it get?

The CodeWarrior debugger is a leading-edge tool, however. As such, it has some remarkable features you would not expect to find in an ordinary debugger. The most interesting and immediately useful cool feature is *watchpoints*. In addition, you can use the CodeWarrior debugger for some pretty advanced tricks, such as multilanguage and multitask debugging.

Working with Watchpoints

Watchpoints are similar to breakpoints in principle, but very different in their execution. Where a breakpoint stops at a particular line of code, a watchpoint stops if the contents of a particular place in memory

change, regardless of what piece of code causes the change to happen. When a new value appears at the location you specify, no matter where the code is that causes the change, the debugger interrupts program flow, and you have caught the code that changes the location in question.

Because a watchpoint is memory-dependent rather than code-dependent, it makes an excellent bug-hunting tool when the problem is some bit of code changing a value it shouldn't be touching. These kinds of bugs can be extremely hard to find, because you have no idea where to start looking. With a watchpoint on the memory location of interest, the debugger can help you find the problem quickly.

NOTE Watchpoint support is still under development in the Windows-hosted debugger, so what you see may be a little different from what I describe here.

Setting and Clearing a Watchpoint

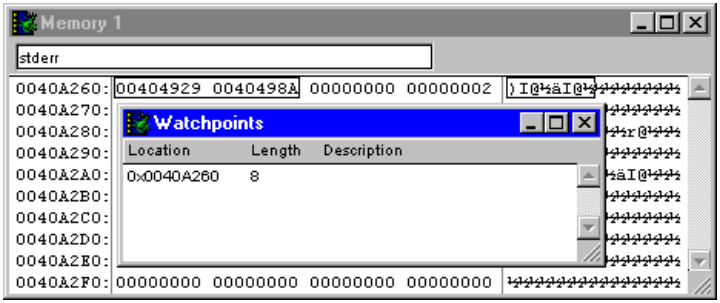
You can watch the value of a global variable, or the contents of an arbitrary range in memory. To set a watchpoint on a global variable, simply select the variable in any debugger view that displays the variable, such as the Globals pane of the Source Browser window, or the Expression window. After selecting a variable, choose the Set Watchpoint item from the debugger's Data menu, and a watchpoint is set for that location in memory.

NOTE You can't set watchpoints on stack-based local variables, or on a processor register.

You can also watch a range of memory. To set such a watchpoint, you need to use the View Memory command in the debugger's Data menu to open a view of a range of memory. Figure 8.29 shows such a view.

FIGURE 8.29

You can set a watchpoint on a range of memory using the Memory view.



You can decide where the view begins by entering the address in the edit field at the top of the window. Select a range of memory in the window, and set the watchpoint on that memory. If something happens in the program to modify these memory locations, the debugger alerts you and stops execution at that point.

Program flow stops immediately. You can then explore where you are and what's going on, so you can determine what code modified that particular location. Chances are it's a perfectly valid operation. Then again, it might be some bit of code overstepping its bounds. In that case, you can stomp the bug.

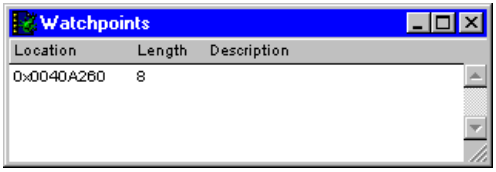
You can get rid of a watchpoint using one or the other of two alternatives. If program execution has stopped as a result of a watchpoint being triggered, choose the Clear Current Watchpoint item from the Data menu to eliminate that particular watchpoint. If you want to get rid of any other watchpoint, select the variable or memory location that has the watchpoint (in whatever window happens to display the watchpoint), and choose the Clear Watchpoint item from the Data menu.

Viewing Watchpoints

As with breakpoints, the debugger has a view that lets you see all your watchpoints at once. To see this window, choose the Show Watchpoints item from the debugger's Window menu. Figure 8.30 shows

what this window looks like. It lists each watchpoint, the number of bytes being watched, and, if it's a variable, the name of the variable. You can also clear a watchpoint from this window. Simply select it and press the Delete key.

FIGURE 8.30
The Watchpoints window lists all current watchpoints.



Multilanguage Debugging

The CodeWarrior debugger steps seamlessly from one language to another. If you write a Pascal function that calls some code in the standard C library, and you've included that code in your project, you can step into the C function just as if you were working in C all along. The same holds true for calls to or from Java.

The language is essentially irrelevant to the debugger. Certain features apply to certain languages, but you can step through your code regardless of the language you use.

Multitask Debugging

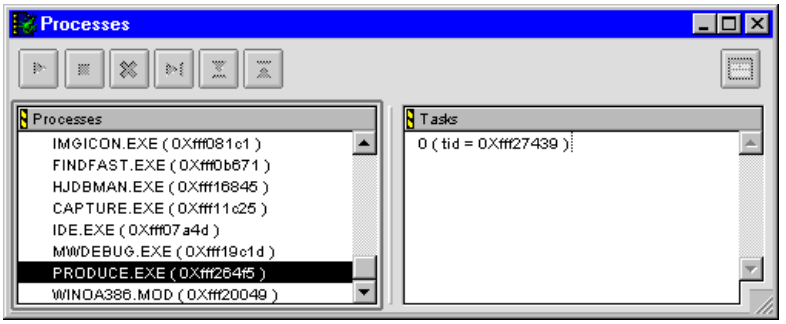
The CodeWarrior debugger supports environments in which multiple tasks may run simultaneously. A good example of multiple tasks are separate threads of execution running under Windows 95/NT, Java, or Mac OS. You can stop each task, explore, switch tasks, and so forth.

To select a particular task or thread, use the Processes window. Choose the Show Processes command from the debugger's Window

menu. In the resulting window, shown in Figure 8.31, every running process appears on the left side. On the right is every task under control of that process.

FIGURE 8.31

The Processes window lets you choose which task to debug when an application has more than one.



Each thread or task can have its own Program window, complete with its own calling chain. To open a window for the task, double-click the task listed in the Processes window. When you do that, everything works exactly as you would expect. You can walk through code, set breakpoints, and so forth. If the operating system supports preemptive multitasking, you can stop one task cold while others continue to function. If the operating system supports only cooperative multitasking, then stopping one task effectively stops them all, because the stopped task cannot yield to other tasks.

Multiproject Debugging

All through this chapter we have been talking about debugging a single project. That's the common model. You build an executable, and you debug it. You build a library, and you debug it. However, the software you're working on may be more complex. You might have a set of modules that work together, but with each being based on a separate CodeWarrior project.

The debugger handles this problem just fine. You can open multiple symbolics files simultaneously. For example, let's say you're building an application that uses three separate dynamically linked libraries. You want to test how all the parts work as a whole. You can use the debugger to open the symbolics files for each of the DLLs, and then run the application under the debugger. You can then use the debugger to analyze all your code. It doesn't matter that the code came from different projects.

These advanced features are very cool. You're most likely to use watchpoints, and perhaps multiproject debugging. The leading-edge features such as support for multiprocessor and multitask debugging may also come in handy, but before you can use them you must be fluent in basic debugger skills. So let's get some practical experience working with the CodeWarrior debugger.

Hands-On Exercises

In this exercise you'll explore the debugger's functionality. This is not an exercise in which you actually hunt down and find performance, logic, or coding bugs in your code. As a programmer you already know how to do that. What this exercise emphasizes is how to get the job done using the Metrowerks debugger.

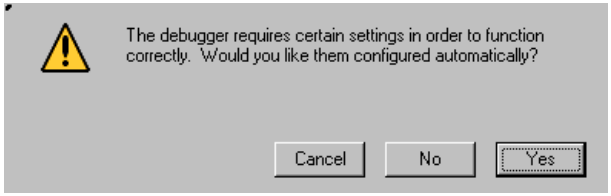
Along the way you're going to walk through a tiny little program that builds random sentences out of a collection of nouns, adjectives, verbs, and other parts of speech. It's designed to show off a variety of debugger features as simply and clearly as possible. The program generates random sentences you might find enjoyable, and along the way you're going to see how to navigate through code, study and modify variables, use breakpoints, and more. So without further ado, let's get started.

1. Launch CodeWarrior and open the project. If CodeWarrior is not already running, launch CodeWarrior, and then open the Sen-

tence.cwp project file from the exercise folder for this chapter, Exer08. Once the project is open you can continue with the next step where you enable debugging.

2. Enable debugging in the IDE. In this step you learn how to set up a project so you can successfully debug it at a later time. To begin, look in the debug column at the right of the main.c file, and notice that it is empty. Then choose the Enable Debugger item from the Project menu. You will get a dialog box, shown here in Figure 8.32, alerting you to the fact that project settings change as a result. Click the Yes button to accept the changes.

FIGURE 8.32
Enabling the debugger is the first step toward creating debuggable code.


The debugger requires certain settings in order to function correctly. Would you like them configured automatically?
Cancel
No
Yes

Next, go to the Project Settings and examine two panels: x86 CodeGen and x86 Linker. You use these panels to decide which format you want to use for debugging. For this exercise, turn on both CodeView and SYM formats in the x86 CodeGen panel so the compiler will create debugging information in each format. Then go to the x86 Linker panel to make sure that the linker generates both a SYM file and CodeView information.

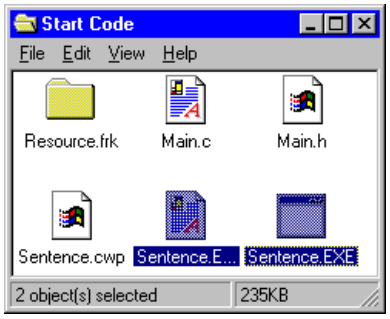
When you have ensured that all four of these options (two in each panel) are on, you're ready to continue.

3. Make the project. Now that debugging is enabled, you put the compiler and linker to work. Choose Make from the Project menu, and the IDE builds your executable. At the same time, the linker also generates a SYM file, in this case named Sentence.EXE.SYM, after the application. Figure 8.33 shows you the Windows desktop folder

containing the project file, and the new executable and SYM files. The .EXE file contains CodeView, and the .SYM file contains—you guessed it—SYM information.

FIGURE 8.33

The IDE builds both the .EXE and the .SYM files at the same time and places them with your project.



4. Launch the debugger and open the symbolics file. In this step you get started with real debugging. Locate and launch the Metrowerks debugger. When you do, the standard dialog box appears to open a file, and at this point you should open Sentence.EXE as shown in Figure 8.34. You could choose Sentence.EXE.SYM, because either format will work for all the remaining steps in this exercise. However, CodeView is the standard for Windows programming, so we'll stick with it for the time being. (If you target Mac OS you'll want to use SYM.)

When you open the file, you'll witness a sudden flurry of activity as the debugger does three things. First, it opens the browser window that gives you access to all the symbolics—that is, your source code. Next, it automatically launches the application. The Log window for this process appears, listing each DLL that has been loaded for this application. Then, when the launch process is complete, the debugger's Program window appears in front of everything else. Figure 8.35 shows the Program window, and on display is the **main()** function of the main.c file. Execution is stopped at the beginning of **main()**, as indicated by the current statement arrow.

FIGURE 8.34

Open the Sentence.EXE file to access the CodeView symbolics.

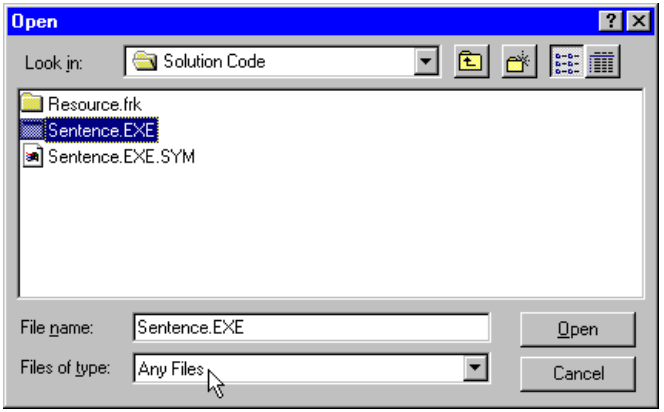
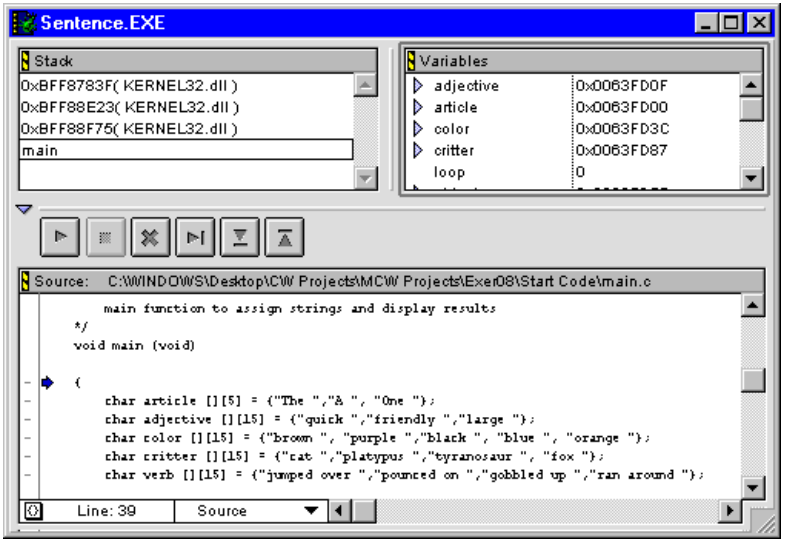


FIGURE 8.35

The Program window is where you control the program running under the debugger.


Sentence.EXE
Stack
0xBFF8783F(KERNEL32.dll)
0xBFF88E23(KERNEL32.dll)
0xBFF88F75(KERNEL32.dll)
main
Variables
adjective 0x0063FD0F
article 0x0063FD00
color 0x0063FD3C
critter 0x0063FD87
loop 0
Source: C:\WINDOWS\Desktop\CW Projects\MCW Projects\Exer08\Start Code\main.c
main function to assign strings and display results
*/
void main (void)
{
char article [][5] = {"The ","A ", "One "};
char adjective [][15] = {"quick ","friendly ","large "};
char color [][15] = {"brown ", "purple ","black ", "blue ", "orange "};
char critter [][15] = {"cat ","platypus ","tyranosaur ", "fox "};
char verb [][15] = {"jumped over ","pounced on ","gobbled up ","ran around "};
Line: 39
Source

NOTE

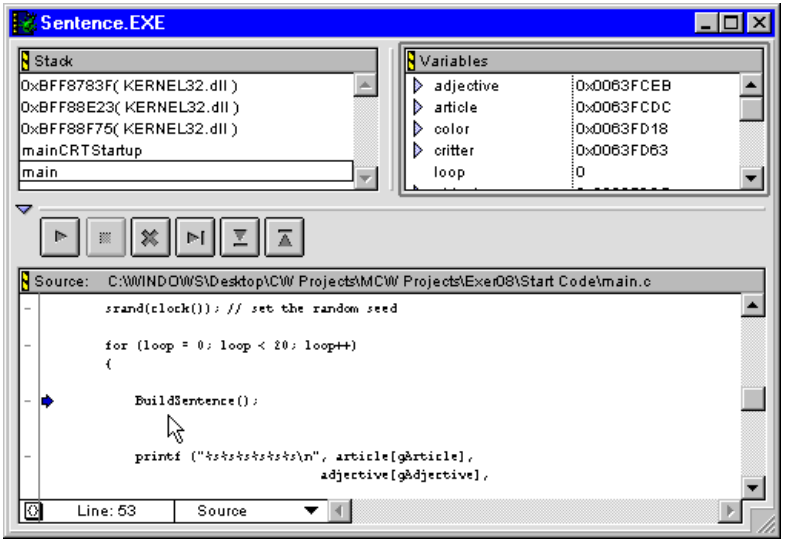
One more window appears as well, but it may be concealed completely by the Debugger MDI window—that's the application's own window, in this case a simple C-console window. We'll take a look at that later.

5. Step through the code. In this step you master the "walk-through" commands you use to step through code. To begin, issue a Step Over command (Ctrl+S) and observe the statement arrow as you do. One line of code executes and execution stops at the next line, ready for your next command.

Continue to issue a series of Step Over commands (a total of about nine) until you reach a line of code that calls the function **BuildSentence()**. Figure 8.36 shows you this place in the Program window.

FIGURE 8.36

Step through code until you reach the call to **BuildSentence()**.

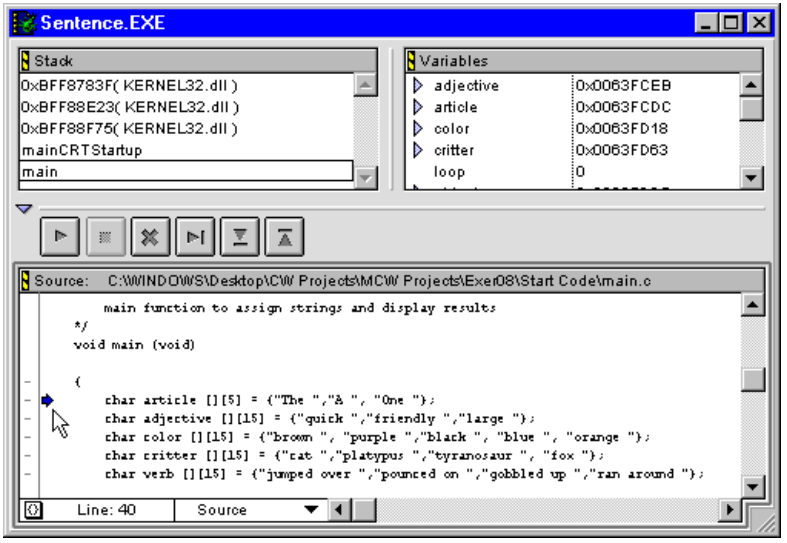


Admittedly, that was a trifle tedious. Let's do it again, a better way.

6. Reset the current statement arrow. Click on the current statement arrow and drag it all the way back to the beginning of the **main()** function. It's nice to be able to go back and start all over without having to quit the application. Figure 8.37 shows the arrow back at the beginning of the **main()** function.

FIGURE 8.37

Move the current statement arrow back to the beginning of the program.

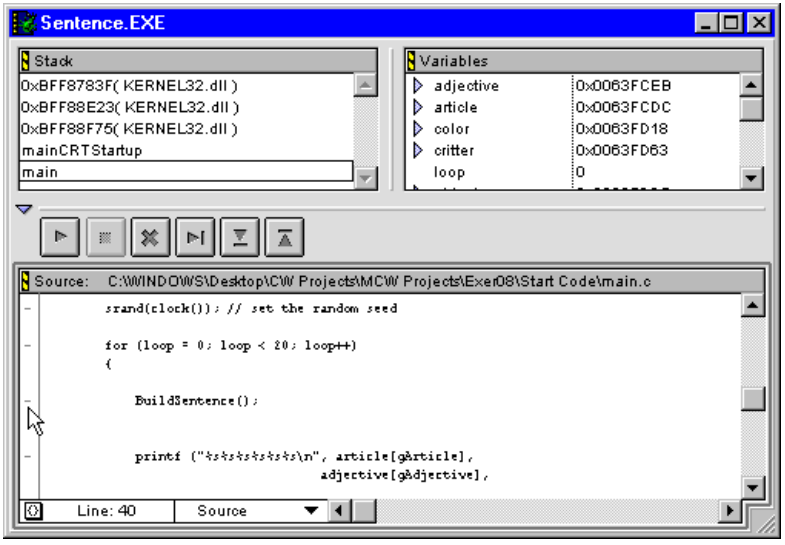


7. Use a temporary breakpoint. Because the place where you want to go is **BuildSentence()** scroll to that location if necessary to display that function call. Instead of manually stepping to get here, press the Alt key and click the breakpoint dash next to this line of code. Figure 8.38 shows this spot. Watch carefully, and you'll see a breakpoint marker flash temporarily, and then the current statement arrow appears. All the code in between has run once again, but you stopped right where you wanted to be.

Before you start the next step, take a look at the Variables pane, just to see the names of variables that are present. These are all the local variables used in the **main()** function. Also observe the contents of the Stack pane (the calling chain) to see what it looks like. Both of these panes are about to change.

8. Step into a function. Issue a Step Into (Ctrl+T) command to move into the **BuildSentence()** function. Figure 8.39 shows the Program window with the current statement arrow inside this function. You have just traced into this function.

FIGURE 8.38
Use a temporary breakpoint to move quickly to a new location.



Take a look at the Variables pane and you'll notice that this function has no local variables. The Stack pane reflects the fact that you are inside the **BuildSentence()** function, and that you came from the **main()** function.

9. Look back up the calling chain. Without stepping through code, just go to the Stack pane and select the **main()** function, and watch what happens to the Variables pane, as shown in Figure 8.40. All the local variables for **main()** now appear, even though you're actually executing inside **BuildSentence()**.

In addition, notice that the current statement arrow is pointing to the line of code right after the function call you just executed. This is not really the current statement, because you're actually elsewhere in code. This is where you will be *after* you return from wherever you are.

WARNING

If you move this arrow, it becomes the true current statement arrow! You leave the subroutine and jump directly back to here without executing any intervening code.

FIGURE 8.39

Stepping into a function shows you its code and its variables, and updates the calling chain.

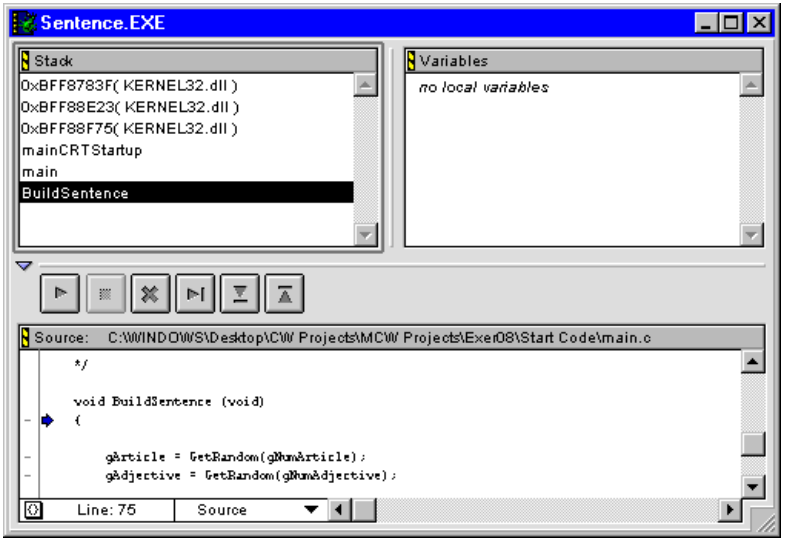
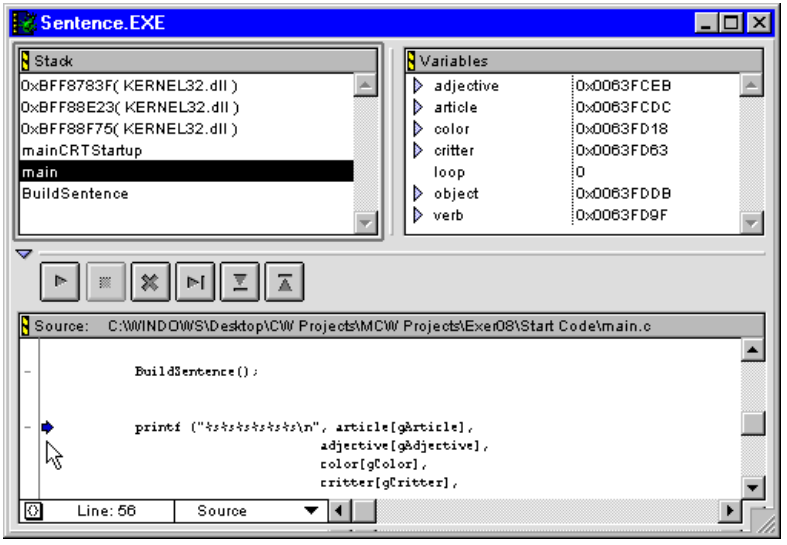


FIGURE 8.40

Use the Stack pane to select the function whose variables you'd like to examine.



To finish this step, use the Stack pane again and select the **BuildSentence()** routine to return to where the code is really executing. You have successfully navigated the calling chain.

10. Examine global variables in the Source Browser. Now that you are back inside **BuildSentence()**, you'll notice it makes repeated calls to a function named **GetRandom()**. For each call, it passes a limiting value (so the random-number function returns a result in the right range for that particular part of the whole sentence), and stores the return value in a global variable such as **gArticle** or **gAdjective**.

Take a look at the Variables pane, and you'll see that no variables appear there. You can see global variables in the browser window. Locate the Source Browser window and activate it. If you use the Window menu, it's the window with the very long name—the entire path to the symbolics file it represents.

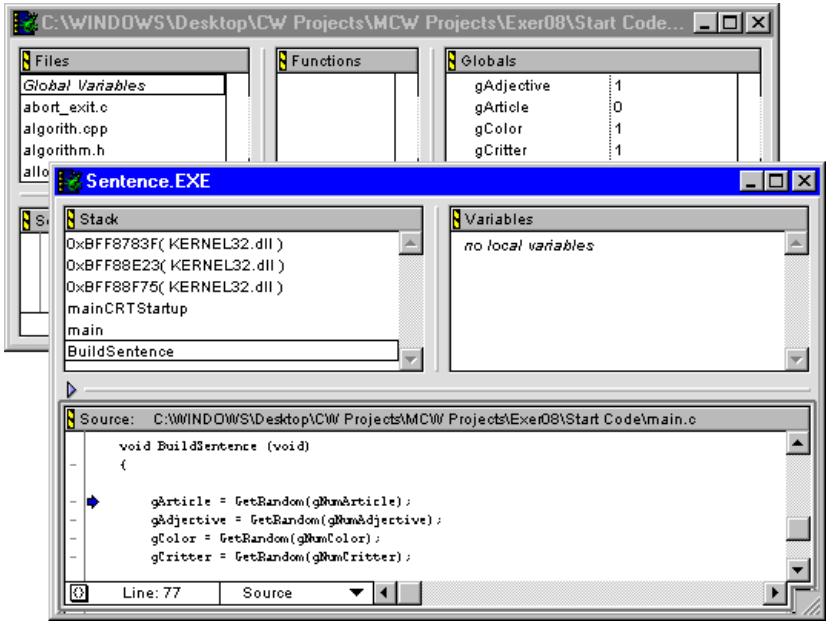
Once you have activated the window, use the Files pane to locate and select the Global Variables item (it's at the top of the list). After you select that item, scroll the Globals pane to locate the variables that begin with a lowercase "g." Finally, position this window so that when the Program window is in the foreground, it doesn't obscure the Globals pane, because you'll want to watch the values change. Figure 8.41 shows such an arrangement.

Now, take a step through this function and watch how the corresponding value changes in the global variables. Each number represents an element in the corresponding array of words from which the final sentence is built. If you wanted to, you could double-click a value and change it. Of course you'd have to make sure the value you entered falls within the legal range for that particular part of the sentence, or you might encounter unexpected results.

In the Program window's Source pane, click the source-code viewing pop-up menu and switch to assembly language. There's your code in its raw, machine-level glory. There's also a current statement arrow.

FIGURE 8.41

Set up the debugger's principal windows to view global variables.



Step again, and everything works just like it did in source view. Switch back to source view and step one more time; everything still works just fine.

11. Set and use a breakpoint. In this step you set two breakpoints, one in the Program window and the other in the Source Browser window. First, while you're in the Program window and looking at the code in **BuildSentence()**, set a breakpoint at any one of the calls to **GetRandom()** that you have already passed by. Click the dash to the left of the line and watch the breakpoint appear.

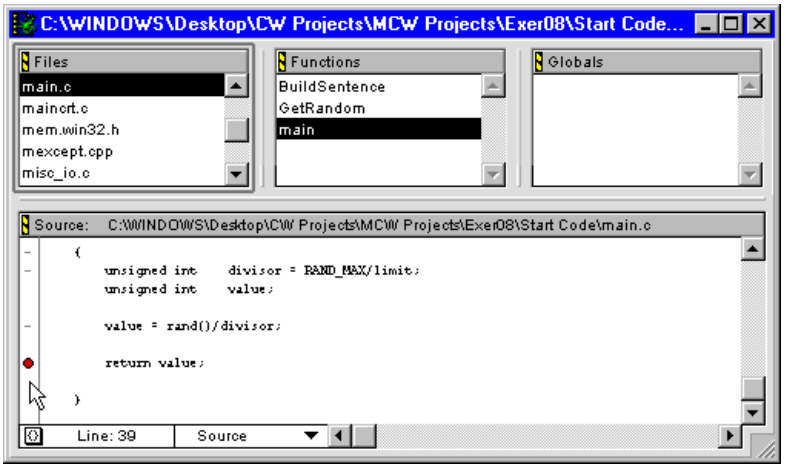
Now switch back to the Source Browser window, and in the Files pane locate and select the main.c file. Use the file pop-up menu at the lower left-hand corner of the window to locate and jump to the **GetRandom()** function, or select the function from the Functions pane in

the top center of the browser window. When you have found the function, set a breakpoint at the line of code that says

return value;

Figure 8.42 shows this spot, with the breakpoint established.

FIGURE 8.42
Set a breakpoint to control where you stop.



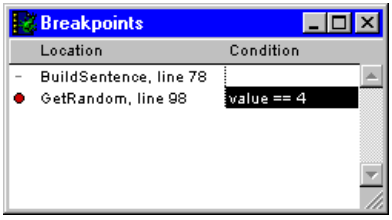
After you have set the breakpoint, resume the application by clicking the Run button (Ctrl+R). Execution proceeds and stops inside the **GetRandom()** function right at your breakpoint. The line of code (the **return** statement) has not executed yet.

12. Set a conditional breakpoint and use the Breakpoints window. If you look around you in the Program window, you'll see that this routine has some local variables, one of which is named **value**. This is the random number the function has created. In this step you're going to use the Breakpoints window to make this a conditional breakpoint, and to deactivate an existing breakpoint.

Choose the Show Breakpoints item from the debugger's Window menu to display the breakpoints window shown in Figure 8.43. Notice that both your breakpoints are listed. To the right of the **GetRandom** breakpoint, double click the conditional column and enter the condition **value == 4**. Then click the breakpoint marker to the left of the **BuildSentence** breakpoint. This deactivates the breakpoint, but you could turn it on again at any time.

FIGURE 8.43

Use the Breakpoints window to control all of a projects breakpoints and set conditions.



Double-click one of the breakpoint listings, and you'll jump right to that code in the Source Browser view. This is a great little navigational aid when you want to see the code right around a breakpoint.

Now run the code again, and in a moment (unless the odds are against you) you should break inside **GetRandom()**. Examine the contents of the variable **value** in the Variables pane, and note that it is indeed four as required by the condition you attached to the breakpoint. This happens to represent the word for the color orange in the building sentence.

TIP

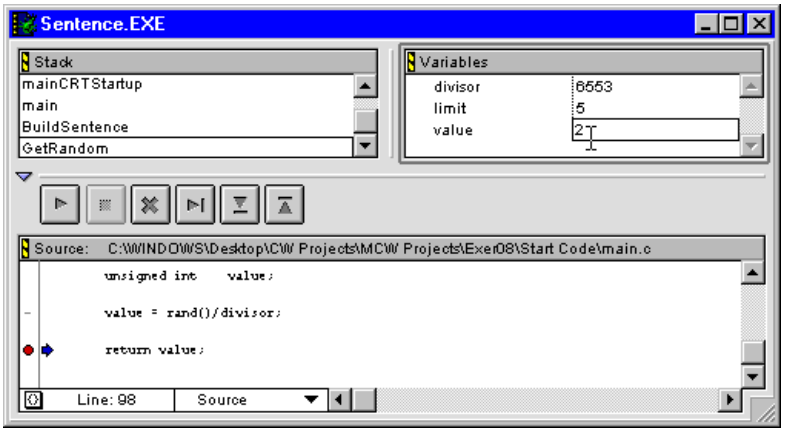
By the way, feel free to shrink the debugger and take a peek at the console window at any time so you can see the words appearing as the program runs. If you watch after a break, the next sentence will have the word "orange" in it.

Next time the program breaks at this spot, double-click the value of **value** so you can modify it, as shown in Figure 8.44. Enter a new value

from zero to three, and then watch what happens when the sentence appears. This time the color isn't orange, because you changed the value.

FIGURE 8.44

You can change the value of a variable in the Variables pane.



NOTE

If by chance the program does not break (the odds are about 1 in 100), then the program builds all twenty sentences without using the word "orange." In this case, switch to the actual console window that displays the sentences, press the Enter key (which exits the program), return to the debugger, and run the program again.

All right! You have set, cleared, used, and conditionalized a breakpoint, used a temporary breakpoint, and studied and modified local variables. Lets explore ways to look at variables a little more closely.

13. Modify variable display type. For this step, navigate to the **main()** function in whatever way you see fit. Possibilities include the following:

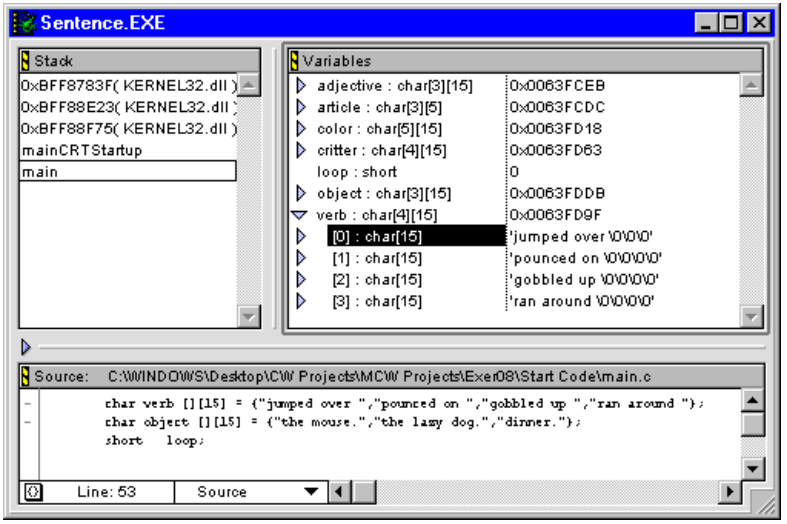
- Going to the Source Browser window and setting a temporary breakpoint inside the function
- Stepping out of functions until you return to **main()**

- Using the call chain to jump to the code and then using a temporary breakpoint in the Program window

However you get there, once you arrive take a look at the Variables pane and scroll down to the very end so you can see the variable named **verb**. Now go to the Data menu and choose the Show Types item so the debugger shows data types. You'll see that **verb** is a two-dimensional array, and the debugger tells you how big it is. Because **verb** is actually a char*, it is a hierarchical variable—it has a disclosure triangle to the left of it. Click the triangle to display its contents, and you'll see that it displays an array of values, as shown in Figure 8.45.

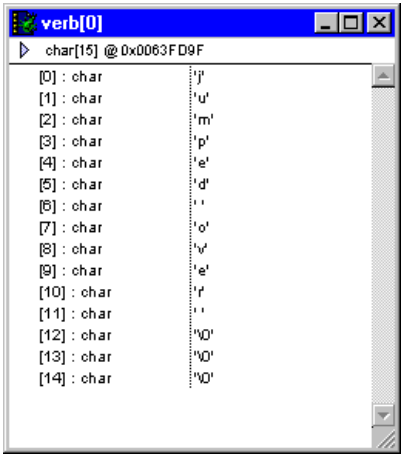
FIGURE 8.45

You can see data types and contents of any variable, however complex the hierarchy.



Now select one of the strings inside **verb**, as shown in Figure 8.45, go to the Data menu again, and choose to view this as a C string rather than as an array of characters. All the meaningless slashes and zeroes disappear, and the data becomes much easier to read. Take a moment to explore the various data display options you have. Because these are arrays, you can also open an Array window for **verb**, or any of its contents. Figure 8.46 shows such a window.

FIGURE 8.46
Use the Array window to easily access any element of an array.



If you resume execution and subsequently return to the **main()** function, all the work you have done disclosing or setting up variables is lost. In the next step you use the Expression window to solve this problem.

14. Use the Expression window to watch variables. Use the Show Expressions item in the Window menu to display the Expression window. Once you do, you can gather variables from anywhere you can find them. Possible sources include the following:

- The globals pane in the Source Browser window
- The variables pane in the Program window
- The code in any Source pane
- Any variable on display in any debugger window

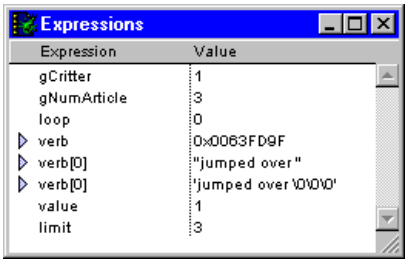
Once you've spotted a variable, the next step is to get it into the expressions window. To do this you have two tricks. You can:

- Select the variable and choose Copy to Expression (Ctrl+D) from the Data menu
- Drag the variable from its location to the Expression window and drop it there

Figure 8.47 shows an Expression window with a few variables placed inside. Take some time to experiment with the Expression window—modify variable values, change data display, drag elements of the window around to reorganize them, select and delete items, add new items, and so forth.

FIGURE 8.47

Use the Expression window to gather data from all over the code.



Most important of all, walk through your code and watch what happens to the contents of the Expression window as you enter and leave functions. Variables that go out of scope have blank values, but remain present and accounted for, and are automatically refreshed when they come back in scope. This is not unlike the functionality of the messaging window architecture in the IDE—disparate information from a wide variety of locations gathered into one place for your efficient and practical use.

The debugger offers more features and flexibility than we can investigate in this brief introduction, but we've covered the core and all of the most important features. I encourage you to start exploring some other uncharted territory. You may recall that at the beginning of this exercise you created a SYM. You might try opening the SYM file and seeing how it works compared to the CodeView information we have been using—you won't see any difference. Beyond that, however, you'll find a lot of room for experimentation.

For example, we haven't touched on the Processes window, the Memory window, or the Register window. The debugger is full of nooks and crannies and paths to intriguing places. It is likely that you won't use some of these features very often, but it's nice to know they're there if and when you need them. Spend some time familiarizing yourself with the full breadth and depth of the debugger. The debugger—and the time you spend exploring it—will serve you well.

Summary

You have mastered the fundamentals of the CodeWarrior debugger. You can launch it, locate symbolic information, use the debugger to navigate anywhere in your code, set and control breakpoints, use conditional breakpoints, display data in any format you wish, wander through memory, see and modify processor registers, work in a high-level language or assembly language, establish watchpoints, and much, much more. Wow!

As a parting comment for the debugger, let me pass on one serious recommendation. Despite all that you have read and done, there is still more to the debugger. For the debugger, and for all the other CodeWarrior tools, you should examine the extensive CodeWarrior documentation for additional details on features we have discussed, and for other features we haven't covered. The tools are changing all the time, adding new tricks and enhancing old functionality. The documentation keeps up with the latest developments, and has a good deal of reference material not included here. Metrowerks has over 7,500 pages of documentation on its various tools, products, and technologies. The good news is, you don't have to read it all! Focus on those parts that interest you most. Combined with the experience you're gaining here, you'll be in great shape.

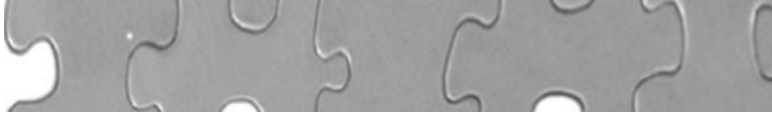
Speaking of experience, you have reached a significant and major milestone. This marks the end of Part III of *Mastering CodeWarrior for Windows 95/NT*. You have met and explored all the core tools in the CodeWarrior IDE. Congratulations!

What remains in Parts IV and V are topics that explore the real purpose of CodeWarrior—writing world-class code. You want to use these tools to write real code for real targets, and that's what we're going to start talking about next.

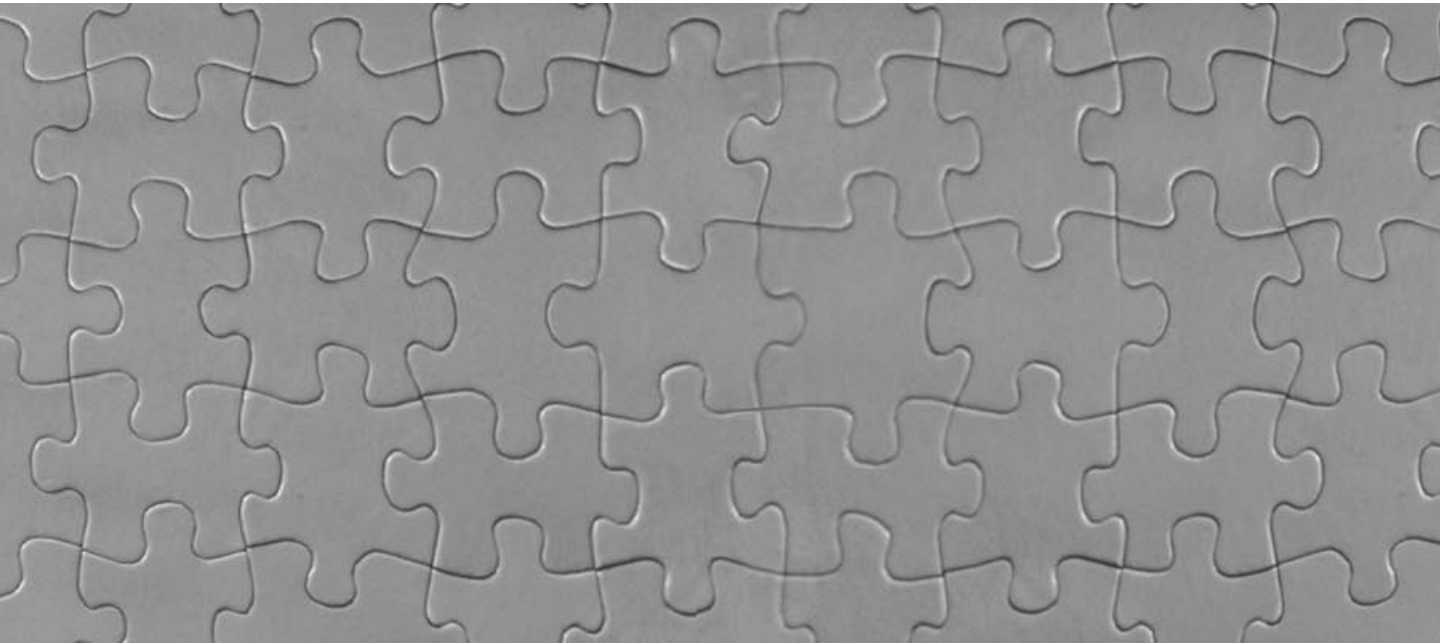
Part IV covers the libraries and frameworks that come with CodeWarrior, and the tools you use with them. CodeWarrior is a treasure-trove that includes thousands and thousands of lines of reusable code for each and every major target. This section of the book introduces you to that code.

Part V discusses the specifics of writing code for three principal targets—Windows, Java, and Mac OS. When it comes time to actually create a project for a particular target, you will encounter intriguing twists, and you may meet entirely new tools. You'll learn about runtime libraries, debugging for a different platform, how to run Java applets, and a whole lot more.

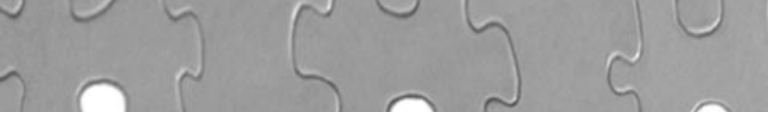
We have a ways to go on our journey, but it's safe to say you have reached a real peak. Whole worlds of programming for all kinds of targets are in view.



PART



IV



Working with Libraries and Frameworks

No modern development environment is complete unless it provides you with some significant code to get you on your way. Part IV introduces you to the code libraries and application frameworks that come with CodeWarrior.

After completing this part of the book, you'll know Constructor inside and out. You will also be familiar with the libraries and frameworks in CodeWarrior; you'll know where they are and how to use them. This book won't make you a true master of any of these frameworks, because an application framework is a complex beast. Explaining the intricacies involved is well beyond the scope of this book. However, you will know what awesome code is available to you as a code warrior.

CHAPTER 9

NINE

Using Standard Libraries

- Using the Standard C Library
- Using the Standard C++ Library
- Using the Pascal Library
- Cool Features
- Hands-On Exercises

CodeWarrior comes with three complete, robust, platform-independent code libraries—Metrowerks Standard Library (MSL) C, MSL C++, and Pascal. These libraries will work on any appropriate target for which the language is supported in CodeWarrior. All three of these libraries work equally well for Windows 95/NT or Mac OS.

The MSL and Pascal libraries provide you with everything you would expect in a library—all the required routines defined in the language standard. In this chapter you'll examine each of these three libraries. For each library you'll see the kinds of routines that are available, and learn how to incorporate the library into your own code.

If you're a Pascal programmer, don't skip the discussion of MSL C, because in the "Cool Features" section we'll talk about how you can use MSL C directly from your Pascal code.

The Fundamentals of Standard Libraries

At the dawn of the desktop computer era, high-level languages were platform-specific. If you wrote code in Applesoft BASIC on your Apple II, no matter how generic your code was, you were out of luck if you wanted to run that code on some other operating system.

Subsequently, programmers put major effort into creating platform-independent languages. With such a language (Pascal and C/C++ both qualify) you can write generic code that should—in theory at least—work under any operating system. The goal of this effort was to create cross-platform compatibility—the endless quest of all programmers.

continued ➠

The Fundamentals of Standard Libraries

To accomplish this goal, language designers had to overcome a significant barrier. Many of the things a typical program wants to do require platform-specific code! Different processors have very different architectures, and different operating systems have very different demands. For example, how do you allocate memory inside the RAM space of a particular platform? The details are sure to vary in critical ways.

Faced with this conundrum—addressing varying operating system architectures with a single language—the language designers made an important decision. They decided to keep the core of the language extremely simple. The language would concern itself with data types, logic, and program flow (branch statements). All critical, platform-dependent activities—such as memory management, disk access, even math functions and string manipulation—would be excluded from the language.

What use would such a language be? Not much, except for the other half of the design equation. Each of the common, platform-specific areas of concern was defined to exist in a standard library. In effect, the designers broke the language into two pieces—a core piece of platform-independent concepts, and a standard library of routines to address platform-dependent concerns. Then, for any given platform, the same core applied, but a platform-specific library would be provided. Code that called only these library functions would then work on any platform.

The library is a formally defined collection of code. It includes a precise set of routines with specific and mandatory parameters and return values. Every standard implementation of the library on any operating system is guaranteed to have these functions. The language is incomplete and effectively useless without the standard library.

continued ➠

The Fundamentals of Standard Libraries

Unfortunately, at the time that C and Pascal came into existence, the graphical user interface (GUI) as we know it did not exist. So the definition of standard behavior was limited to those tasks performed with a command-line interface and a character-based display. There was no support for any GUI feature, however trivial. The libraries still know nothing of windows, mice, scroll bars, buttons, and so forth.

As a result, if you limit your programming to pure language and the standard library, you can write platform-independent code—and this is very cool. Of course, complications always seem to arise, but in principle and in fact it can be done. However, if your code has a user interface, the end result will be a primitive, lowest-common-denominator application that fails to take advantage of the GUI features available today. If what you're writing has no interface (such as a calculation engine of some kind that generates data for later use), then you can write world-class, cross-platform code using just the libraries.

Using the Standard C Library

You typically want to answer only two important questions when dealing with a library of code: what's in it, and how do you use it. If you are an experienced programmer who has used standard C libraries in the past, then you will be right at home.

NOTE

The American National Standards Institute (ANSI) has completely defined the C language and its library—they finished the task in 1989. The Metrowerks C library fully implements the ANSI standard.

Understanding the C Library

The C library is a collection of functions devoted to a variety of general purposes. This section does not go into detail on each and every function—the library contains about 200 of them after all. The *MSL C Reference* that is part of the CodeWarrior documentation fully describes each function.

TIP

The Metrowerks library reference documentation—a separate manual is available for each library—is particularly good. It not only describes the technical details of each routine such as the prototype, arguments, and the files you must include to use the function, in many cases the manuals present solid example code showing you how to use the routine effectively. This is especially true of the often-problematic input/output functions.

Instead of taking a detailed look at individual routines, this section gives you a high-level view of what's in the library. If you know the kinds of things you'll find there, and you know where to find out more, then you'll be well on your way toward mastering the C library.

The functions available in MSL C cover the following principal categories:

- **General utilities**—declared in stdlib.h, for memory management, random numbers, string-to-number conversions, searching sorted data, and so forth; includes functions such as **malloc()**, **free()**, **rand()**, **atof()**, **atoi()**, **strtod()**, and **bsearch()**
- **Input/output**—declared in stdio.h, for creating, naming, opening, closing, writing, and accessing data in files, and for working with character-based input from a keyboard; includes functions such as **fopen()**, **fread()**, **fwrite()**, **fclose()**, **printf()**, **scanf()**, **getchar()**, and **putchar()**
- **Characters**—declared in ctype.h, for identifying different kinds of characters (such as numbers, punctuation, or uppercase) and converting between upper- and lowercase; includes functions

such as **isalpha()**, **isdigit()**, **islower()**, **ispunct()**, **tolower()**, and **toupper()**

- **String manipulation**—declared in string.h, for copying, moving, concatenating, and comparing strings; includes functions such as **memcmp()**, **memcpy()**, **strcat()**, **strcmp()**, **strcpy()**, and **strlen()**
- **Floating-point mathematics**—declared in math.h, for trigonometry functions, logs, modulo arithmetic, roots and powers, and so forth; includes functions such as **sin()**, **cos()**, **tan()**, **atan()**, **atan2()**, **log()**, **log10()**, **fmod()**, **pow()**, **exp()**, and **sqrt()**
- **Time**—declared in time.h, for getting and calculating time-related values; includes functions such as **clock()**, **difftime()**, **gmtime()**, and **localtime()**

These six categories account for perhaps 80 to 90 percent of all the library functions. Other functions are available for less common purposes, such as support for Unix-style commands.

With the routines in the standard library you can accomplish a great deal, as demonstrated in this chapter's hands-on exercises. As you can see, the functionality of the library centers around number and character manipulation. Numbers and letters (in the form of strings of text) account for the vast majority of content in most computer programs—either directly, or by providing the underlying data for graphical representations.

Using the C Library in Your Projects

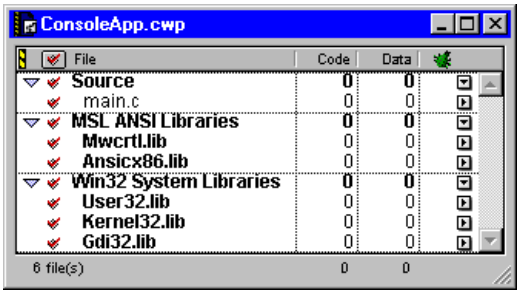
Using the C library in any particular project is extremely easy. In fact, all of the projects you have worked on so far for this book have included and used it. Nevertheless, you can still make choices in how you include the library in your project and gain access to its functions.

Most stationery projects that use the C language include MSL C automatically. Fundamentally, the standard library is part of the language, and while it is possible to write C programs without it, that doesn't happen very often. Figure 9.1 shows you the stationery file for

a simple console application using the C language. Other stationery files include this library as well.

FIGURE 9.1

Using stationery is the easiest way to add the MSL C library to a new project.



As you can see, the stationery takes advantage of the fact that the simplest, most common, and most trouble-free method for gaining access to the library code is to add the compiled library to your project. The name of the file is Ansicx86.lib, and you'll find it in the CodeWarrior\Metrowerks Standard Library\MSL C directory. To add this library manually, use the Add Files item in the Project menu and navigate to this directory. This same directory also includes the actual CodeWarrior projects and source code used to build the compiled library.

NOTE

Actually, different versions of the library are available for different targets. The MSL C PPC.lib file is designed for Mac OS running on PowerPC. The MSL C.68k(4i).lib is for Mac OS running on a 680x0 chip. The "4i" refers to the length of an integer (4 bytes) that is a user-selectable option in 68K programming, and the library you use must match your compiler settings in this regard. You must use the proper version of the library for your target, but starting with stationery takes care of this detail for you.

At the same time you should add Mwcrtl.lib—the Metrowerks C runtime library—found in the CodeWarrior\Win32-x86 Support\Libraries directory. The code in this library handles start-up chores,

and provides background support for a variety of routines such as exception handling in C++ code, You shouldn't have to use the functions in the runtime library directly, but you probably won't be able to run your code without it.

TIP

You can find the source files for the runtime library (and the CodeWarrior project that builds the actual compiled library) in the CodeWarrior\Win32-x86 Support\Libraries\(Runtime Support) directory.

Adding the library is only half the task—that gives the linker access to the routines in the library so it can use the code. However, for the compiler to understand your code it must know what you're talking about. Whenever you call a standard library function, the file in which you make the call must include the header that declares the function. The Metrowerks documentation details which header file you must include for every function in the C library.

If you are making extensive use of the C library, you should consider creating a precompiled header for your project that includes the standard library headers. You can then include that precompiled header in your project, via the project prefix or by using **#include** statements in the files that need the header information. Doing so saves substantial time when you rebuild your project.

TIP

You don't have to include the .pch file (which creates the precompiled header) in the project if you don't want to. If you know the underlying header files are not going to change, you can just include the precompiled header and not the .pch source file.

With MSL C you have one more option when it comes to adding the library to a project. Metrowerks includes the actual library source code so you can add the source files to your project if you want to.

We'll talk about this approach in the Cool Features section, because it applies to both the C and C++ libraries equally.

Using the Standard C++ Library

An ANSI committee is currently working on the C++ language, but its task is not complete so the language is still only partially defined. The standard is a moving target emerging slowly out of the standards committee. As a result, the Metrowerks C++ library is also something of a moving target. Metrowerks tracks the emerging standard and keeps the MSL C++ as current as possible.

WARNING

Keeping the C++ library abreast of the ANSI standard has a downside as well. When the standard changes (it has in the past and may again in the future until the language is fully defined), the library changes. As a result, code you wrote using an earlier version of the library might break with a new version. The alternative is to fall behind the leading edge, which is not a practical option. If it is important not to break your code, you can continue to use an older version of the library. However, keeping current with standards is vital in the long run.

In this section you'll follow the same path as you did for the C library—you'll look at what's inside the library, and then at how to use it.

Understanding the C++ Library

The C++ library is not for the faint of heart. It is an advanced, state-of-the-art library with a heavy reliance on template classes, including the Standard Template Library (STL) developed by Alexander Stepanov as it has been included in the formal draft standard.

The C++ standard library provides a flexible framework you can modify and extend to meet your specific needs. As we did with the C library, let's look at the categories of classes you have among the rich

and powerful code available to you in the library. The *MSL C++ Reference* covers the nitty-gritty details. The general categories or areas in which the library provides support include the following:

- **Language support**—common type definitions, numeric limits (largest and smallest numbers and so forth), and support for runtime services such as exception handling and runtime type identification
- **Diagnostics**—a full set of extensible classes to support exception handling
- **General utilities**—support for other library classes, including dynamic memory allocation, and templatized support for standard mathematical and logic operators including member functions such as **plus()**, **minus()**, **times()**, **divides()**, **logical_and()**, **logical_or()**, and so forth
- **Strings**—string classes provided for manipulating text represented as sequences of type **char**, or any other character-like data type, including member functions such as **append()**, **assign()**, **insert()**, **erase()**, **copy()**, **swap()**, and **replace()** as well as string search functions
- **Localization**—a set of classes that provide support for representing data in various formats particular to different countries, including monetary values as well as date and time data
- **Containers**—standard template classes for a wide variety of data containers such as lists, stacks, vectors, queues, maps, and so forth, with member functions to insert, remove, replace, iterate, search, and compare elements in the container
- **Iterators**—a group of standard template classes that implement five different kinds of iterators: input, output, forward, bidirectional, and random access
- **Algorithms**—a series of standard template classes that provide algorithms for searching and performing numerical operations
- **Numerics**—support for complex number types and numeric arrays

- **Input/output**—stream support is the primary mechanism for data input and output in a C++ program, and it can be used with other elements of the library, particularly strings, locales, and iterators

You can spend a lot of time getting your head around the intricacies of the C++ standard library, but it does provide a powerful framework of generic, interchangeable building blocks you can use with your own data types.

TIP

If you want to pursue programming with the Standard Template Library, consider the *STL Tutorial and Reference Guide* by David R. Musser and Atul Saini (Addison Wesley, 1996).

Using the C++ Library in Your Projects

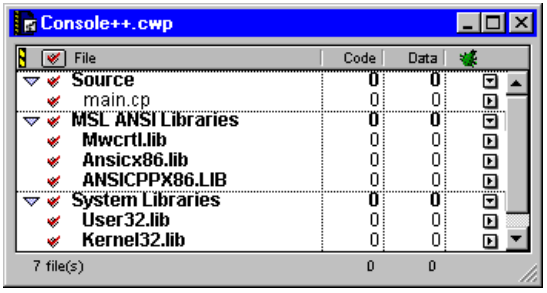
Once again, stationery is the easiest way to go when you want to use this library in a new project. Stationery that involves the C++ language includes the MSL C++ library for you, as shown in Figure 9.2. The compiled library is named Ansicppx86.lib.

Adding the library manually works exactly as does the C library. You can

- Include the actual source code, or

FIGURE 9.2

A C++ console application includes the MSL C++ library as well as MSL C.



- Add the compiled library directly to your project and
  - include particular header files as necessary or
  - create a precompiled header for the entire library and include that

The source code for MSL C++, the compiled libraries, and the CodeWarrior projects that build the libraries, are all located in the CodeWarrior\Metrowerks Standard Library\MSL C++ directory. The precise names of the compiled library files and their targets are:

- ANSICPPX86.LIB—Windows and x86 programming
- MSL C++ PPC.lib—Mac OS on PowerPC
- MSC C++.68K.lib—Mac OS on 680x0

As with MSL C, the C++ library comes in source-code format so you can step into that code when debugging and see exactly what is going on. This capability can be a tremendous asset when you are learning how to use the library.

If you use the C++ library, you must also use MSL C as well. The C++ library relies on the presence of the C library, and builds on it. Do not try to use the C++ library with a standard C library from some other vendor—it won't work. While these libraries are "standard," that term really applies only to their external appearance. That is, each of the libraries includes the same public functions with the same arguments and return values, but the internals can vary substantially.

WARNING

The MSL C++ includes the STL because it is now part of the C++ language standard. However, the STL was originally and still is a separate entity, and you may have the STL as a product from some other source. If you do, make sure you do not include any of the headers from that STL at the same time as you include headers from the MSL C++, because they will clash with each other. You will end up with many compile-time errors as a result of multiple declarations of terms.

Using the Pascal Library

Pascal as a language does not have the same clear distinction between core language and standard library that the C language has. In Pascal, library functions and procedures are considered a built-in part of the language. As a result, you don't need to declare them or include them in any way—they are predefined by the language itself.

However, different programming environments implement Pascal in proprietary ways. Lots of overlap exists, to be sure, but some differences exist as well. As a result, Pascal code is not quite as portable from one compiler to another the way strict ANSI-C code usually is.

Understanding the Pascal Library

The Pascal library consists of about 80 routines and procedures that cover the usual range of services you expect in a standard code library. The Metrowerks Pascal library is compatible with Turbo Pascal.

Once again, let's take a high-level look at the kinds of routines you have available, and leave the details to the Metrowerks reference documentation—in this case the *Pascal Library Reference*. As with the other library documentation, this manual gives you not only chapter and verse on each routine, but often provides example code as well.

If you have read the preceding discussions about MSL C and MSL C++, the categories of routines in the Pascal library will look very familiar. The bottom line is that most programs on most platforms in most languages all have to do the same kinds of things, so the similarities between libraries far outweigh the differences. The Pascal library includes the following categories of routines:

- **Memory management**—routines to handle the allocation and release of dynamic memory, such as **New()**, **Dispose()**, **Release()**, **SizeOf()**, **HeapResult()**, and **MemAvail()**

- **Input/output**—routines for creating, naming, opening, closing, writing, and accessing data in files, and for dealing with text input and output, such as **Open()**, **Close()**, **Get()**, **Put()**, **Read()**, **Write()**, **Seek()**, **ReadLn()**, and **WriteLn()**
- **String manipulation**—routines for copying, moving, concatenating, and working with strings such as **Concat()**, **Copy()**, **Delete()**, **Insert()**, and **Length()**
- **Floating-point mathematics**—routines for trigonometry and number manipulation, such as **Sin()**, **Cos()**, **Sqr()**, **Sqrt()**, **Exp()**, **Ln()**—natural logarithm, **Round()**, and **Trunc()**
- **Ordinal operations**—routines to decrement and increment numbers, convert a number to a character, convert a character to uppercase, and perform other simple transformations on numbers, such as **Dec()**, **Inc()**, **Chr()**, **UpCase()**, **Odd()**, and **Ord()**
- **Bit manipulation**—routines to test and modify individual bits in a value, such as **BAND()**, **BNOT()**, **BOR()**, **BXOR()**, **BRotL()**, **BRotR()**, **HiWrd()**, and **LoWrd()**
- **Byte and array manipulation**—routines to work with arrays of bytes, such as **FillChar()**, **MoveLeft()**, **MoveRight()**, and **ScanEq()**
- **Termination routines**—routines to terminate a block or the entire program, **Exit()** and **Halt()**, respectively

The routines in these categories represent virtually all the services available in the Pascal library. The list above does not include every routine in each category, but does represent a fair sampling of the kinds of routines available. See the *Pascal Library Reference* for full details on what's available.

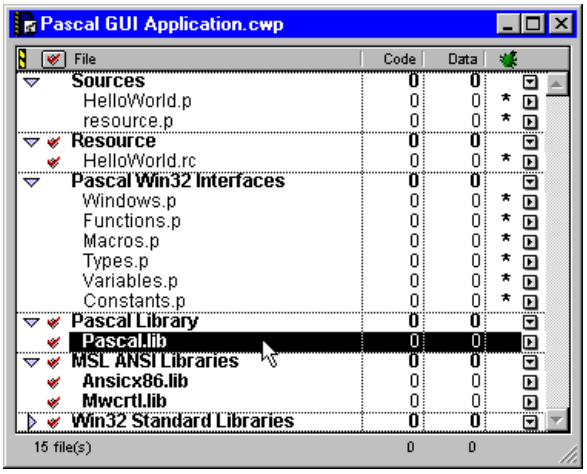
Using the Pascal Library in Your Projects

To use the Pascal library, you simply include the Pascal.lib file in your project. You'll find this compiled library in the CodeWarrior\Win32-x86 Support\Libraries directory. Pascal projects also include MSL C

and the runtime library as well. If you start with Pascal stationery, this has been taken care of for you. Figure 9.3 shows you the stationery project for a Windows application based on Pascal.

FIGURE 9.3

Pascal-related stationery includes all the necessary libraries.



For all practical purposes, you can treat the routines in the Pascal library as if they were a part of the language, because they are. Simply call the routines as and when you will in your Pascal code.

The only issue that might arise is name collision. For example, what happens if you define or use a routine that has the same name as one of the library routines? The locally-defined routine takes precedence—that is, when you call such a routine, your routine gets called, not the library routine.

You can still access the library routine by adding the **System.** qualifier to the call. For example, if you have your own version of the routine **Concat()**, and for some reason you want to call the library routine rather than your own version, you would write the call as **System.Concat()**, passing the correct parameters of course. This feature lets you

override default language routines when you wish, and still have access to the language when you want it.

Cool Features

You'll find two fundamentally cool things about the way Metrowerks deals with libraries. You get the actual source code with each library, and you can use the C library from Pascal easily. Let's look at each of these features in a bit more detail.

Using Library Source Code

As mentioned before (and worth mentioning again), the Metrowerks standard C and C++ libraries come with complete source code. Having the actual source code is a tremendous benefit for several reasons.

For example, because you have the actual source code you can easily debug library calls. Not that you should find many bugs in the library itself, but you can see how the values you pass into library routines are being used. This capability can highlight problems in your own code far more quickly than the alternative. Without source code, the library is essentially a black box into which you pass values and get back results. If the results are not what you expect, you have no way of finding out why that might be—and the bug hunt becomes much harder.

Being able to see library code is also a great learning tool if you're fairly new to programming. Here you have a wide variety of core routines, written by seasoned programmers and tested by thousands upon thousands of your fellow developers. You can see how others have solved similar problems, and you can learn from their work.

Having source code also means you can recompile the library to suit unique and specific needs. If you want to compile with particular compiler settings, or omit certain files you know are of no interest, you can do so. You can, for example, create your own custom libraries

based on the source base. Once again, without the source code, this task is impossible.

And—for those who like to live dangerously—having the source code lets you tweak the code. You can modify the underlying source code and recompile a new library, although such a course does not come highly recommended. All of the libraries have gone through extensive and formal validation. When you start changing things, you run the risk of introducing bugs—so be *very* careful.

Calling MSL C from Pascal

The C standard library is a bit more comprehensive than the Pascal library—it has more functions that do more things. Metrowerks makes it easy to call virtually any MSL C function directly from Pascal, using the same parameters you would if you were calling the function directly from C source code.

To accomplish this neat trick, Metrowerks provides Pascal interfaces for most of the standard C library. In your program's USES clause, you specify certain Pascal interfaces that correspond to the C-language header files. You also include the interface files in your project. After that, you can call the C functions just as if they were in the Pascal language.

TIP

The standard Pascal stationery does this for you by including the file Functions.p among the Pascal interfaces. Take a moment to examine that file using the CodeWarrior editor, and right near the start of the file you'll find most MSL C functions declared.

A few limitations exist, however. Certain functions in the C library are not supported in Pascal—that is, no Pascal interface corresponds to the C library routines. Most notable are the **printf()** and **scanf()** functions (and related routines), and the routines defined in the math.h or

float.h files. Pascal already has most of the functionality represented in math.h and float.h.

The only other factor to keep in mind is that if you use string routines in MSL C, you must pass C-style null-terminated strings and not Pascal-style strings that have a size byte preceding the string. The C library expects C-style strings. With these few limitations in mind, you are free to include the C code directly in your Pascal projects. And of course, as you debug your code you can step right from one language to another without a problem.

Hands-On Exercises

In this exercise you observe a project that uses the MSL C library extensively. The particular application in question calculates and displays the dates and types of the next twenty solar eclipses after a given date, and it does so using MSL C exclusively. A couple of routines perform all the calculations, but the actual mathematical work relies on underlying standard library functions such as **sin()** and **cos()**.

1. Launch CodeWarrior and open the project. If CodeWarrior is not already running, launch CodeWarrior, and then open the Eclipse.cwp project file from the Start Code folder for this chapter, inside the Exer09 folder.

2. Add libraries. Although you could easily build such a project from the appropriate stationery and avoid this task, it is important to understand how to add a library to a project because you may want to use your own custom libraries at some future time. Besides, it's extremely simple.

The existing project has a group titled MSL ANSI Libraries, and the Mwcrtl.lib library (the runtime library) already exists in that group. In this step you add the MSL C library to the same group, alongside Mwcrtl.lib.

First, select the group title MSL ANSI Libraries so that when you add the library it appears in this group. Then choose the Add Files item from the Project menu. In the resulting dialog box, navigate to the CodeWarrior\Metrowerks Standard Library\MSL C directory and add the Ansicx86.lib file. Figure 9.4 shows this dialog box with that file selected.

FIGURE 9.4

Add the Metrowerks standard C library to the project.

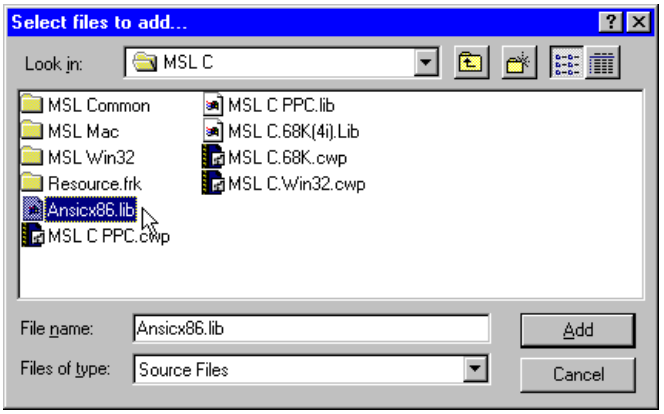


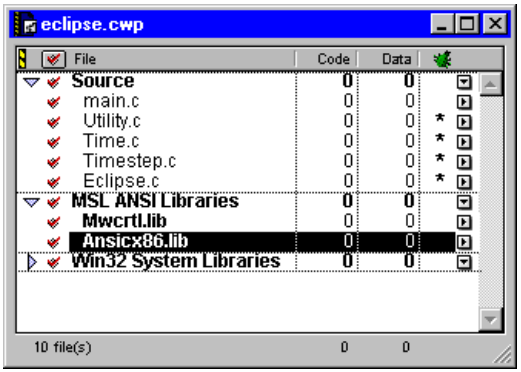
Figure 9.5 shows you the project window after you have added the library. The Ansicx86.lib is a compiled library—it was created by another CodeWarrior project—and it already contains all the object code generated by the library.

Having completed step 2, the library code is now available for the linker to include in your project.

3. Include the library headers. In addition to adding the library, the source code in the project must also include the appropriate header files so the compiler knows about the functions in the library that are used by the code. In this step you add **#include** statements to two project files, main.c and luminary.h.

FIGURE 9.5

The Eclipse.cwp project now includes MSL C.



Open the main.c file, and scroll to the beginning of the file if necessary. At that spot you'll see the following code:

```
// Exercise 9 step 3, add these headers
// #include <stdio.h>
// #include <stdlib.h>
```

Uncomment the two lines with **#include** statements so these two lines become active code. Figure 9.6 shows you what the file looks like after you're through. Each of these standard library header files declares prototypes for standard library functions used in main.c. If you attempted to compile this file without these **#include** statements, the compiler would tell you that certain functions had no prototype.

Save the changes you made to main.c, and then open the file luminary.h. This is the project's master header file that declares project-wide data structures and values. Near the beginning of this file you'll see the following code:

```
// Exercise 9 step 3, add this header
// #include <math.h> /* trig and math functions */
```

FIGURE 9.6

Including standard library headers makes the function prototypes available to the compiler.

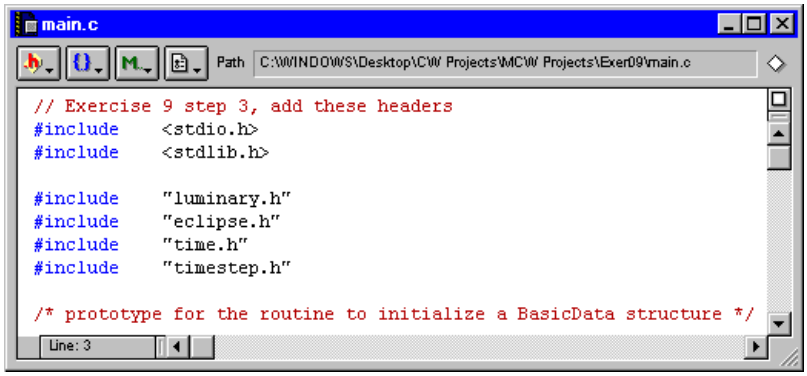
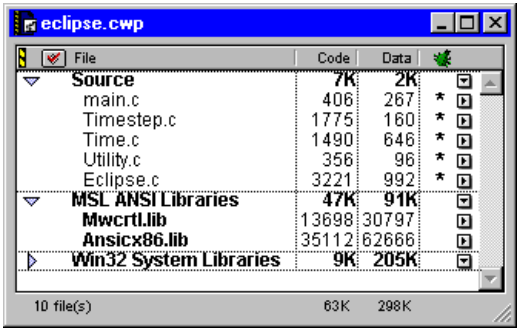


FIGURE 9.7

The eclipse project after a successful make.

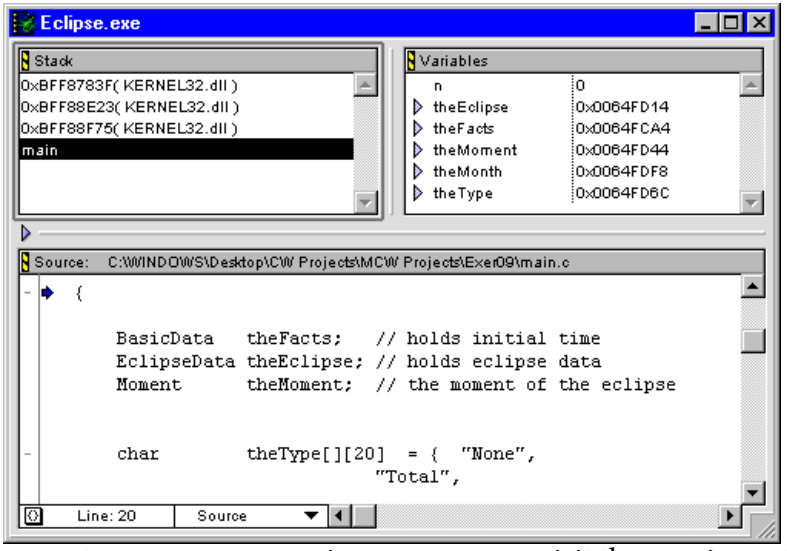


Once again, simply uncomment the line that includes the necessary header file, and save your work.

4. Build the final executable. To compile and link the project, choose the Make command from the Project menu. If you have the toolbar showing, you'll see the progress of the compiler and linker. This project already has debugging turned on, so symbolics information is created as well. When the make is finished, the project window shows code and data for all files in the project, as shown in Figure 9.7.

FIGURE 9.8

The debugger stops at the beginning of **main()**, ready to execute the first line of code.



If you get any error or warning messages, revisit the previous steps to ensure you completed them correctly.

5. Launch the executable under debugger control. In this step you run the application under debugger control. Debugging has already been set up for you for this project—you learned how to do that in the previous chapter.

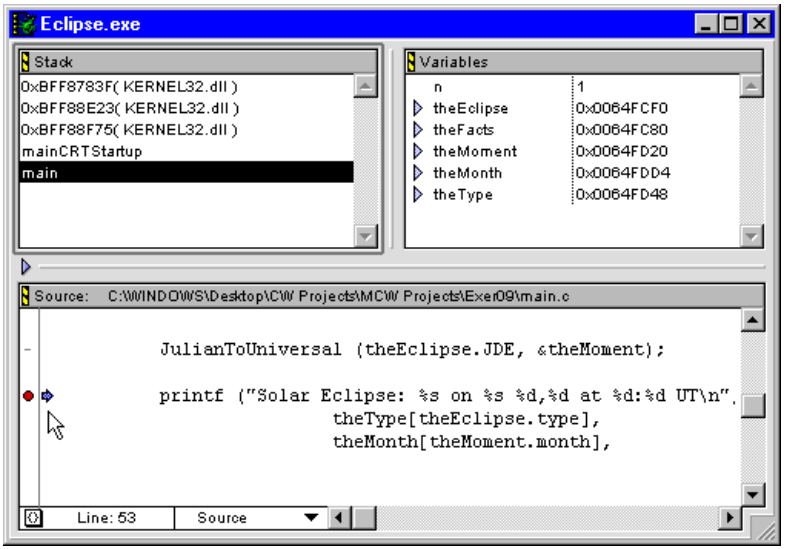
As you know, you have more than one way to launch this program under debugger control, but the simplest is to choose Debug from the Project menu. Do that now—choose Debug. In response, the debugger launches, opens the symbolics information for this executable, launches the application, and stops at the beginning of **main()**. Figure 9.8 shows you the debugger's Program window ready to go.

6. Set a breakpoint and run the program. In a subsequent step in this exercise you'll trace execution into the library source code. To make sure you stop at the right spot, in this step you'll set a breakpoint.

The debugger's Program window is already displaying the correct file, main.c. Scroll down in that file to locate the call to the **printf()**

function, and set a breakpoint at that call. Figure 9.9 shows you the spot with the breakpoint.

FIGURE 9.9
Set a breakpoint at a call to **printf()**.

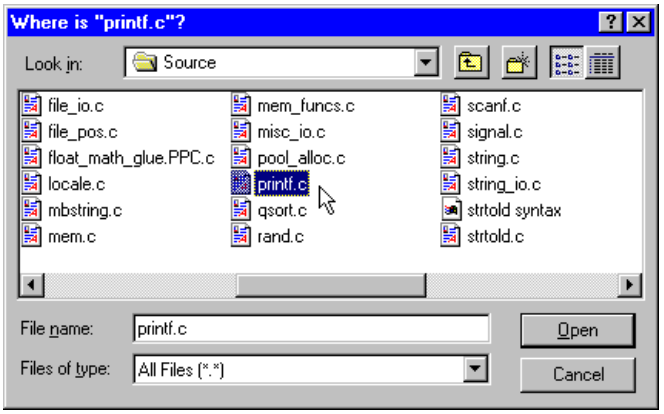


After setting the breakpoint, run the program. It breaks almost instantly at the breakpoint you just set. You are now ready to step into library source code.

7. Trace execution into the standard library. Step into the **printf()** function—you can click a button, use the Control menu, or type Ctrl+T. When you issue this command, the debugger asks you to locate the file printf.c. This source file is not part of the project, and the debugger has no idea where it is. Figure 9.10 shows you the standard dialog box asking you to locate this file. Navigate to the CodeWarrior\Metrowerks Standard Library\MSL C\MSL Common\Source directory and open the printf.c file located there.

FIGURE 9.10

Locate the printf.c file for the debugger.



After you open the file, the debugger displays the source code in the Program window, as shown in Figure 9.11. You can continue to trace execution through the standard library if you wish, but that isn't necessary here. The point is to demonstrate how you can add and use the standard library, even at the source-code level, and you have accomplished that goal.

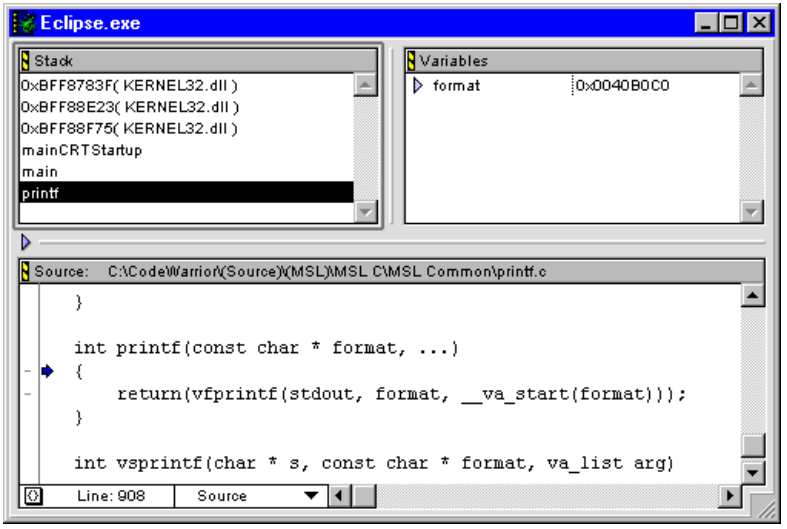
To prepare for the next step in this exercise, step out of the **printf()** function and return to **main()**. Remove the breakpoint at the **printf()** call so it doesn't interfere with program execution.

8. Run the program. In this step you let the program run its course. Issue the run command, and then (if it is hidden) display the console application to see the results of the calculations. The dates, times, and types of twenty solar eclipses are displayed, as shown in Figure 9.12. Press the Enter key to end the program.

The default starting time in the source code—set in the function **InitBasicData()**—is January 1, 1994. Feel free to modify the date to search for other eclipses. The time calculated is the moment in universal time (Greenwich Mean Time) at the peak of the eclipse. This little program does not calculate where on the Earth you need to be standing to see

FIGURE 9.11

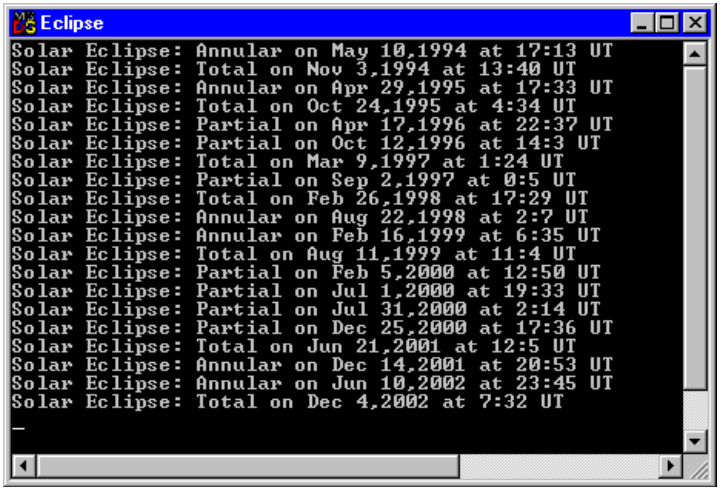
The debugger displays standard library source code for your perusal.



the eclipse, or the time of the eclipse at your location. However, a routine in eclipse.c calculates lunar eclipse data. You might want to try that out as well.

FIGURE 9.12

Eclipse calculates and displays a list of solar eclipses.


Eclipse
Solar Eclipse: Annular on May 10,1994 at 17:13 UT
Solar Eclipse: Total on Nov 3,1994 at 13:40 UT
Solar Eclipse: Annular on Apr 29,1995 at 17:33 UT
Solar Eclipse: Total on Oct 24,1995 at 4:34 UT
Solar Eclipse: Partial on Apr 17,1996 at 22:37 UT
Solar Eclipse: Partial on Oct 12,1996 at 14:3 UT
Solar Eclipse: Total on Mar 9,1997 at 1:24 UT
Solar Eclipse: Partial on Sep 2,1997 at 0:5 UT
Solar Eclipse: Total on Feb 26,1998 at 17:29 UT
Solar Eclipse: Annular on Aug 22,1998 at 2:7 UT
Solar Eclipse: Annular on Feb 16,1999 at 6:35 UT
Solar Eclipse: Total on Aug 11,1999 at 11:4 UT
Solar Eclipse: Partial on Feb 5,2000 at 12:50 UT
Solar Eclipse: Partial on Jul 1,2000 at 19:33 UT
Solar Eclipse: Partial on Jul 31,2000 at 2:14 UT
Solar Eclipse: Partial on Dec 25,2000 at 17:36 UT
Solar Eclipse: Total on Jun 21,2001 at 12:5 UT
Solar Eclipse: Annular on Dec 14,2001 at 20:53 UT
Solar Eclipse: Annular on Jun 10,2002 at 23:45 UT
Solar Eclipse: Total on Dec 4,2002 at 7:32 UT

As always, I encourage you to explore on your own. You might want to add the C library source files to your project and build the project without including the compiled library, and then run under debugger control. When you finish exploring, you can close the project and exit CodeWarrior.

Now that you have completed the exercise for this chapter, let me remind you of a critical (but invisible) aspect of this code, and one of the principal points of this whole chapter. The code you have been playing with in Eclipse.cwp is platform-neutral. You can take this identical source code to a Mac OS computer, a BeBox, or any platform that has a standard C library, compile it, and run it without change. Of course you'll have to add different runtime libraries, and some target-specific version of the standard C library, but the source code itself is 100 percent cross-platform, courtesy of the designers of the C language.

Summary

This chapter introduced you to the standard C and C++ libraries, and the Metrowerks Pascal library, and showed you how to use them in a project. Most important, you know where to go in the CodeWarrior documentation for full details on each and every function available to you in all of these libraries.

As you have learned, these libraries have their benefits and their limitations. The most serious limitation is the lack of real support for a graphical user interface. To solve this problem, object-oriented programmers use application frameworks. And that's what we'll talk about in the next chapter.

CHAPTER 10

TEN

Using Frameworks

- Using the Microsoft Foundation Classes
- Using Java Packages
- Using PowerPlant
- Hands-On Exercises

The world of software development continues to evolve, and computer software has (thankfully) moved beyond the command line. We now have scrolling windows, pull-down menus, icon-based toolbars, point-and-click interfaces, and much more. However, programming for a complex graphical user interface (the good old GUI), is a real challenge. To make the task easier, programmers who specialize in each of the three principal targets this book addresses—Windows, Java, and Mac OS—have developed application frameworks. CodeWarrior comes with a full application framework for each of these targets.

For Windows programmers, the framework of choice is the Microsoft Foundation Classes (MFC). For Java, it is the Abstract Windowing Toolkit (AWT) from Sun MicroSystems, one of a set of class packages provided with Java. For Mac OS programmers, PowerPlant from Metrowerks has emerged as the premiere framework because its elegant design and modular architecture allow you to use various parts of the framework independently without inheriting all the overhead of the complete package.

CodeWarrior, being a tool devoted to cross-platform development, gives you the premiere framework for each target. At all levels, from language and target all the way to application framework, CodeWarrior gives you the power to choose.

Keep in mind, the purpose of this chapter is not to teach you how to program for a particular platform using any of these frameworks. Each framework has been the subject of several full-length books, and deservedly so. Learning a framework is an entirely separate adventure from the journey we are now on. What this chapter can and will do is show you how to use the frameworks that come with CodeWarrior.

The Fundamentals of Application Frameworks

At its core, programming for a modern GUI is significantly more complex than programming for a traditional command line. An event-driven architecture is less predictable. The user can do almost anything at almost any time, thus imposing great demands on your software to be flexible and responsive. In addition, to establish some standardization—at least within a single platform—modern operating systems such as Windows or Mac OS provide most of the programming calls (the application programming interface or API) you need to create basic or default GUI elements such as buttons, menus, windows, and so forth. This means you must master the requisite API in order to program effectively for a particular GUI operating system.

Mastering an API of this complexity is a nontrivial task. The reference material extends for thousands upon thousands of pages. Bookstore shelves are lined with titles on how to program for one or the other of these platforms.

The good news is that almost every single application aimed at a particular GUI uses the same features. And that fact led trailblazing programmers to develop frameworks. The purpose of an *application framework* is to give you a standard, generic, empty application that you can modify to suit your own purpose.

In general, an application framework concerns itself with the interface—what the program looks like—and not what's inside the application—the data content. That's because all programs look alike in the sense that they are built out of a common set of visual elements, but each program has a unique kind of data it works with. Nevertheless, a good framework also provides classes that support common kinds of data structures such as lists, stacks, arrays, and other ways of working with the raw numbers at the heart of a program.

continued ➠

The Fundamentals of Application Frameworks

A framework also provides flow control—it gathers and understands events and calls the appropriate function in response, either in the framework itself or in your own code. You no longer have to worry about where and when things happen—the framework takes care of that. You have to concern yourself with what to do in response. For example, you no longer have to worry about when to draw a window because the framework will call your drawing function any time it's necessary to draw. You just have to figure out how to draw the window.

Common problems such as menu handling and event parsing have already been solved by expert programmers, and you'll find the solutions inside the framework. You not only get to reuse them, you can extend the framework to suit your needs by using object-oriented programming techniques. You can override certain behaviors and replace them with your own, and still use most of the framework unchanged.

If you are an object-oriented programmer who has used a framework, this is old hat. If you have never used a framework, you should consider exploring one. A framework is not an easy thing to learn—for the most part frameworks are large, complex, and intricate. Nevertheless, investing the time in learning a framework pays substantial dividends in the long run, because you can generate better, more reliable code faster than you ever thought possible. The framework frees you from most of the drudgery of GUI programming so you can concentrate on the unique and interesting things in your application. It gives you a generic application empty of content, and effectively takes you half-way to your goal before you write a single line of code—and *that* is very cool.

Using the MFC

Fundamentally, using the MFC in CodeWarrior is no different and no more difficult than using a standard library. The techniques for adding the MFC to a project and calling its functions are exactly the same as those you use for the MSL C or MSL C++.

Using MFC Stationery

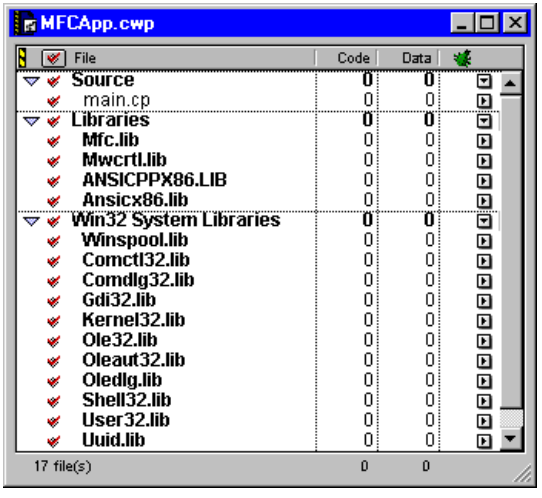
Metrowerks provides an MFC Application stationery project, illustrated in Figure 10.1, that automatically does everything you need. It adds the compiled library for MFC that comes with CodeWarrior—Mfc.lib. It also sets the project prefix in the C++ Compiler panel in the project settings to use the precompiled header MFCHeaders.

TIP

A stationery project also exists for an MFC application with everything set up so you can use the browser with the MFC.

FIGURE 10.1

The MFC Application stationery includes all the necessary files for working with the Microsoft Foundation Classes.



You might take the same approach if you build a project from scratch. The simplest way to incorporate MFC code into your application is to add Mfc.lib to the project. You'll find Mfc.lib in the \CodeWarrior\Win32-x86 Support\Libraries\Mfc directory. The actual CodeWarrior project used to create the Mfc.lib compiled library is in \CodeWarrior\Win32-x86 Support\(MFC4.2).

TIP

If you're building a project from scratch, you must include several Win32 libraries, and the Metrowerks runtime library—Mwcrtl.lib—as well. The MFC Application stationery project takes care of all of this for you, and also includes the MSL (both C and C++).

Including MFC Headers

The only real issue that arises with respect to using the MFC is how to include the header information. The MFC is a very large chunk of highly interdependent code, with many header files. While it is perfectly legal to include headers in the traditional fashion—using **#include** statements as necessary—doing so ensures that your builds take a long time.

A wiser approach is to use a precompiled header for the MFC. You don't even have to create your own. Metrowerks provides both a .pch file and a complete precompiled header for you. The .pch file is named MFCHeaders.pch, and the actual precompiled header is named MFCHeaders. Table 10.1 summarizes the various components of the MFC and where they are located in a typical CodeWarrior installation.

NOTE

As mentioned previously, the MFC Application stationery project also includes the MFCHeaders precompiled header as the project prefix.

TABLE 10.1 Location of MFC files in the \CodeWarrior\Win32-x86 Support directory

Component	File Name	Directory
Compiled library	Mfc.lib	\Libraries\Mfc
Library project	MFC 4.2.cwp	\(MFC4.2)
Source code	n/a	\(MFC4.2)\MFC 4.2 Source
MFC .pch file	MFCHeaders.pch	\(MFC4.2)
Precompiled header	MFCHeaders	\Headers\Precompiled Headers

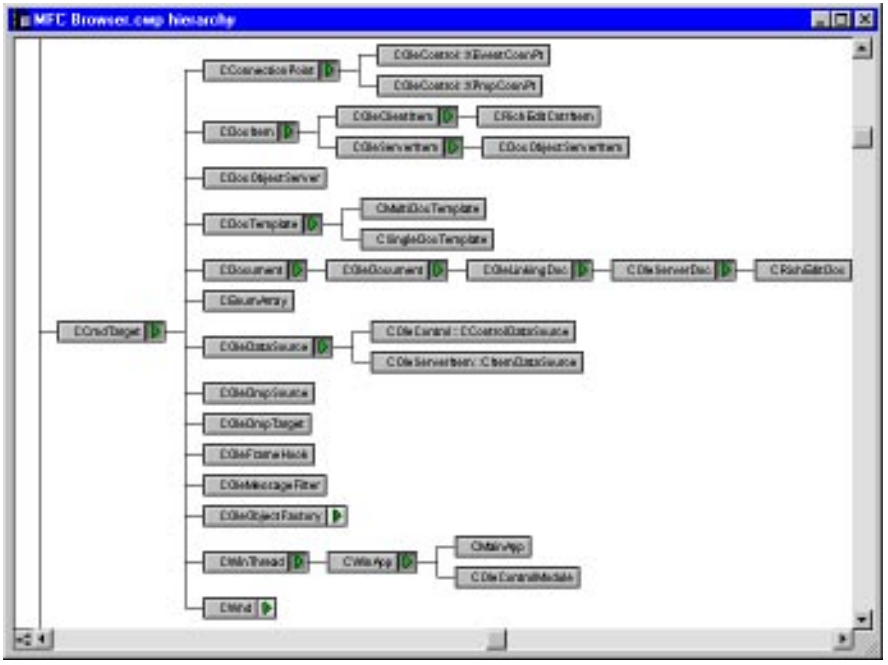
You may want to modify the precompiled header. As you know, you can only include one precompiled header via the C/C++ Compiler panel, and you may want to include more than just the MFC. For example, you may want to add a precompiled header that includes the MFC and the MSL C and C++ headers. To do so, you can copy the Metrowerks-provided .pch file, add the additional header files you want to include, and then create a new precompiled header.

If you want the browser catalog to include MFC classes, you must include the MFC header files in such a way that the compiler sees them. The best way to do this is to add the MFCHeaders.pch file to the project. When the compiler reads the file, it builds the MFCHeaders precompiled header, and encounters all the class information it needs to generate a complete browser catalog. The simplest way to do this is with the MFC Application project that uses the browser. After you build the project, you'll be able to use the browser to see all of the MFC. Figure 10.2 shows you a small portion of the framework on display in the browser's hierarchy view.

Using MFC Source Code

As it does with almost all libraries of code, Metrowerks provides the source code for the MFC. That means you can add MFC source files

FIGURE 10.2
You can use the CodeWarrior browser to navigate anywhere in the MFC.



directly to the project should you wish to do so. This certainly makes debugging convenient as you don't have to tell the debugger where the files are. However, once you have told the debugger where to find one file in a directory, the debugger can find the others in that same directory automatically. Because all of the MFC source files are stored in the same folder, even if you don't include the MFC source files you only have to find one file to find them all. So, to avoid lengthy builds, the best approach to MFC programming is to simply include the library and forget about the source files.

NOTE

Source code may not be available with the CodeWarrior Lite version accompanying this book.

Using Java Packages

Sun Microsystems includes several packages in its Java language software. A package is a collection of related Java classes. Taken together, the packages included with Java form the application programming interface (API) for the Java Virtual Machine (JVM). These are the classes that contain the Java functions you use to write code for the JVM.

You might think that these classes are analogous to the standard library in C or C++, and in a way they are. However, a more accurate analogy would be to say that these Java packages are analogous to the Windows SDK or the Mac OS Toolbox. They amount to the function interface you use to write code for the JVM, and can be considered the equivalent of the Java operating system. The packages are:

- **java.applet**—classes to support applets
- **java.awt**—classes for GUI support
- **java.io**—classes for input and output
- **java.lang**—classes that support the core language
- **java.net**—classes for network support
- **java.util**—classes for various purposes such as stacks, vectors, random numbers, and so forth

When you target the JVM, you use the classes contained in these packages. Full details on Java as a language and the Java packages can be found in the CodeWarrior documentation in a series of HTML-based documents also available directly from Sun. They are the *Java Language Spec* and the *Java API Documentation*. CodeWarrior documentation also includes *Targeting Java*, which explains in detail how to create and manage Java projects in CodeWarrior.

TIP

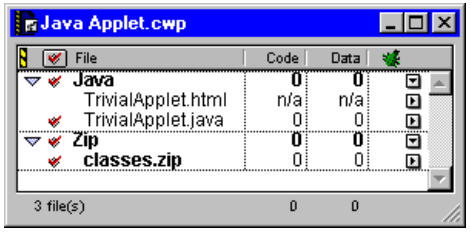
CodeWarrior also comes with the Java Generic Library, which is a set of powerful Java classes from ObjectSpace that extend the Sun API.

Using Java Stationery

Getting started with Java packages is essentially identical to the way you work with the MFC. The simplest route is to start out with Java stationery as the basis for your project, as shown in Figure 10.3. If you take this path, all the necessary files appear in your project, and all the project options are set to appropriate values for Java development automatically. It doesn't get any easier.

FIGURE 10.3

Java stationery sets up your project for Java development automatically.



The key to working with the Java packages in your project is a file named classes.zip. This file is the Java library. It contains all the classes in all the packages listed earlier. As with any compiled library, you can simply add classes.zip to your project manually. You'll discover, however, one very interesting twist in how you can work with the classes.zip library.

Working with the Classes.zip File

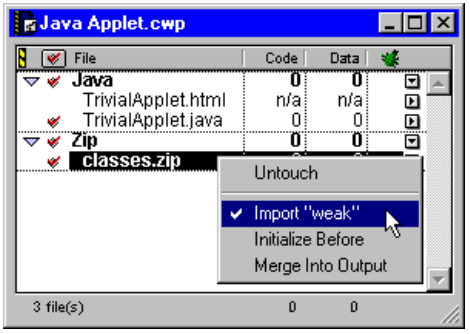
You can think of the classes.zip library as a kind of hybrid or combination of both a DLL and a static compiled library. You control which

way you use the library with the file pop-up menu to the right of the classes.zip name in the project window. Figure 10.4 shows you that in addition to the Untouch command, for this library you have some other choices. They are:

- Import "Weak"
- Initialize Before
- Merge Into Output

FIGURE 10.4

Use the file pop-up menu to decide how to connect your code with the classes.zip file.



If you choose the Import "Weak" option, what you're telling CodeWarrior is that you want to use classes.zip as if it were a DLL. The Java stationery project uses this option. When you use Import "Weak" the compiler will access the information in the library so it can build your object code, but the linker will *not* include the library code directly in your project. In other words, you are importing the library into your code, but only for the purposes of the compile—a weak import.

This decision carries two consequences. First, your final applet is smaller, because it does not include the code from the Java packages. Second, the classes.zip file *must* be available at runtime in a location where your applet can find it, or your applet will not run. This is the identical problem you have when you depend on a DLL. If it isn't

there, your application cannot function. CodeWarrior installs this file in the Windows\Java\Classes directory.

The Merge Into Output command tells CodeWarrior that you want the linker to include the necessary object code from classes.zip directly in your project. This treats classes.zip as if it were a static library, just like the Mfc.lib file we discussed earlier. The consequences of this choice are exactly the opposite of those for Import "Weak"—your applet is bigger, but it doesn't require the presence of classes.zip to run.

NOTE

The linker strips out unused code, so using Merge Into Output does not mean that the entire classes.zip archive gets merged into your project. Only those parts that you actually use are included in the final result.

The Initialize Before command is only useful if you have two mutually dependent libraries. This command tells CodeWarrior which library should be initialized first.

Importing Java Classes

Java does not have header files like C or C++, so you get no opportunity to precompile headers. However, the simple fact of adding the classes.zip library to a project does not give your code access to the functions in the library. You must **import** the appropriate classes using correct Java syntax. This is analogous to **#include** in C and C++. To include everything in the Java API, you would write

```
import java.*;
```

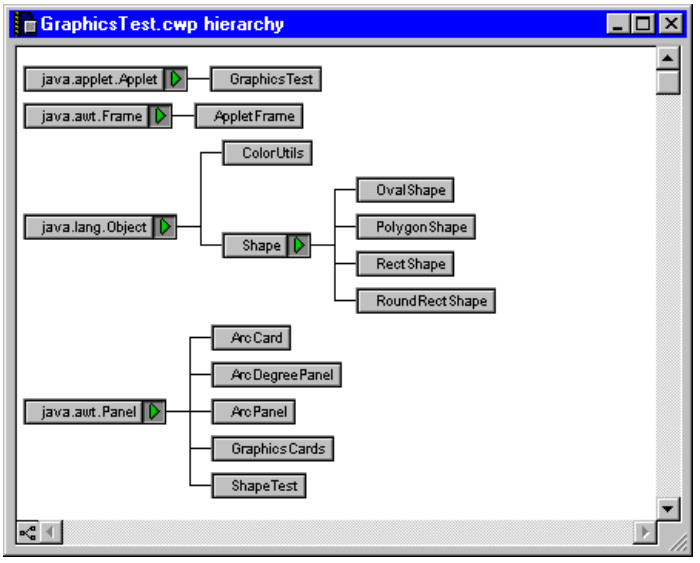
You could be somewhat more precise and import particular packages as necessary for your file, such as

```
import java.applet.*;
import java.awt.*;
```

This approach requires a bit more thought, but focusing on what you really need can make the build process run more quickly because you're importing only necessary information. Whatever you include, you can display in the browser if you have it turned on. Figure 10.5 shows the editor's browser displaying Java class information.

FIGURE 10.5

The browser works for Java as well as for C++.



Using PowerPlant

If you are targeting Mac OS, no better framework is available than PowerPlant, Metrowerks own application framework designed by Gregory Dow. PowerPlant is a modern C++ application framework that is designed to let you use just those parts of the framework that work for you.

If you are targeting the Mac OS, most of the time you will want to use the core classes that represent the visual, command, and control

The Thought behind PowerPlant's Design

A typical application framework uses single inheritance, and as such is something of a monolith. The MFC is a good example—most classes inherit from a single, fundamental class, CObject. The MFC is certainly not alone in this approach—most application frameworks are monolithic in exactly the same way. If you want to use the framework at all, you use the entire thing. Let's call this the "one-size-fits-all" theory of frameworks.

PowerPlant was written from the ground up using a different design principle. PowerPlant's architect, Gregory Dow, wanted to take full advantage of the multiple inheritance features of C++. The end result is a masterpiece of design—one that has been used as the perfect example of the right way to use multiple inheritance.

Rather than one-size-fits-all, PowerPlant is designed as a collection of independent, stand-alone modules that you can mix and match to tailor a custom solution. Each module serves a particular purpose. For example, modules are available to handle visible objects, strings, interobject communication, menus, idle-time processing, and so on.

Within PowerPlant, the modules are joined together by multiple inheritance. For example, visual objects that must send messages multiply inherit from the LBroadcaster class. Objects that must listen for messages inherit from LListener. Basic messaging behavior is *not* part of the entire framework, so classes that do not require this functionality do not inherit this behavior. By contrast, in a typical single-inheritance framework, messaging is usually introduced very early in the hierarchy, and many classes that have no need of the functionality inherit it anyway.

continued ➠

The Thought behind PowerPlant's Design

The multiple-inheritance approach not only keeps classes lean and simple, it also lets you combine different modules to tailor the framework to suit your needs. You can include just those parts of PowerPlant you want to use, and ignore the rest. This makes PowerPlant easier to learn and easier to use because you can focus on those parts that interest you without having to swallow the entire framework at once. Gregory Dow calls this a "buffet" approach to programming—you go to the feast and pick those parts of the meal you enjoy, and no one forces you to eat the lima beans.

Although some PowerPlant hierarchies (such as the visual classes, for example) are effectively dependent on the entire framework, you can use several PowerPlant modules independently should you wish to do so. The messaging architecture makes a perfect example. You can use the LBroadcaster/LListener design as a complete, stand-alone solution for your own messaging needs. Simply declare your object classes so they also inherit from LBroadcaster and LListener, and you can use this solution in your own code. The dozens of other classes and thousands of member functions in PowerPlant disappear and never enter into your project. Similarly, the PowerPlant LString class has powerful member functions for string manipulation and can be used completely independently of the rest of PowerPlant. The implications this has for programming with PowerPlant, and using PowerPlant in your own code, are significant, because you can extract pertinent parts of the code as you need them, not only for Mac OS, but for other platforms as well. Some major pieces of PowerPlant are not dependent on Mac OS.

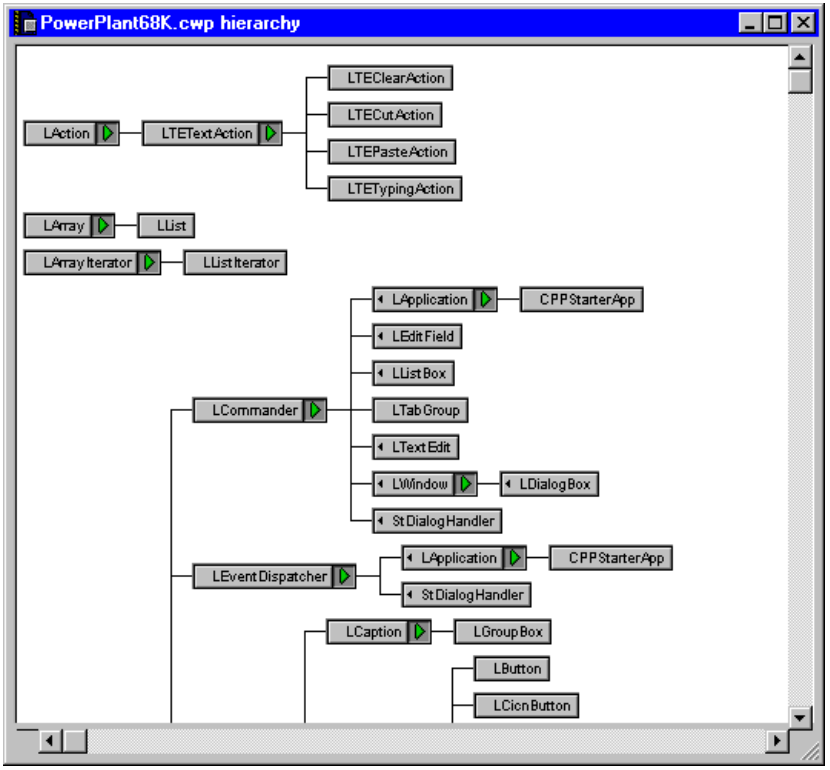
elements of a typical Mac OS application—and indirectly rely on all the core features of PowerPlant. The best approach is to use project

stationery, because it already includes all the files you need—no thought or action required.

You can work manually if you prefer, and add individual files as appropriate. You include header information by using precompiled headers for PowerPlant, or by using traditional **#include** statements where necessary. If you want to see the PowerPlant classes in the browser, you must add the corresponding .pch file to the project, or not use the precompiled header at all. The PowerPlant stationery has the browser turned on and ready to go. After you build a PowerPlant project, you can see the classes on display in the browser, as shown in Figure 10.6. Notice all the little black triangles to the left of class names, indicating classes that have multiple base classes.

FIGURE 10.6

The browser shows you how PowerPlant uses multiple inheritance to implement a modular approach to application frameworks.



One side effect of a design based on multiple inheritance is that the framework (as illustrated in the browser hierarchy) has several relatively shallow hierarchies. Most class hierarchies in PowerPlant run only two or three levels deep. Compare Figure 10.6 with Figure 10.2, which displays part of the MFC hierarchy, and you'll see that a single inheritance hierarchy such as the MFC tends to be much deeper. Having several small trees (as with PowerPlant) instead of one giant redwood can make the forest of classes easier to comprehend.

The one major distinction in the practical details of how you add PowerPlant to a project is that this framework is not included as a compiled static library with CodeWarrior. Where the MFC has the Mfc.lib file you can add to a project, no corresponding .lib file exists for PowerPlant. You typically add PowerPlant to your project by adding source files.

You can build a compiled library yourself should you choose to do so. However, because PowerPlant is a modular framework, you can use individual pieces freely without the risk of having things break. That means that unlike a single-inheritance framework (where you might as well have a library because everyone uses the same one-size-fits-all package), you can tailor PowerPlant to suit your needs. As a result, no single library would suit all programmers and all purposes.

Bottom line, if you are using part of PowerPlant, you don't need to use an entire library. You add just those source and header files necessary for the module you want to use. This makes PowerPlant a real treasure-trove of reusable code.

TIP

Here's the framework "tip-of-the year," especially for Windows programmers. Many of PowerPlant's modules are platform-independent. While the core of the framework is designed for Mac OS programming, several of the modules that make up the entire package have nothing to do with Mac OS. This is particularly true of the support code for such things as arrays, queues, and interobject messaging. You'll discover some world-class code in here, and it is worth a look even if you have no intention of programming for Mac OS.

You'll also discover one more advantage to adding the actual source files to your project. The PowerPlant source files are stored in a directory structure and not all in one folder. If you are debugging PowerPlant code and used a library instead of actual source files, you would have to frequently tell the debugger where to find files—a real annoyance at best. By adding the actual source files to a project, you avoid the hassle. Best of all, because PowerPlant is a lean framework, even compiling it the "old-fashioned" way is very quick.

The Metrowerks documentation on PowerPlant is extensive, and outstanding. *The PowerPlant Book* is the core instructional manual that discusses the framework architecture and how to use the principal classes. *PowerPlant Advanced Topics* covers advanced Mac OS programming and explains how to use PowerPlant for tasks such as implementing undo and redo, drag-and-drop, networking applications, and much more. This documentation also includes a function-by-function reference. Taken together, these documents can bring you up to speed in PowerPlant in less time than you might imagine.

But for now, let's return to the MFC, look at a real project that uses it, and see how it works.

Hands-On Exercises

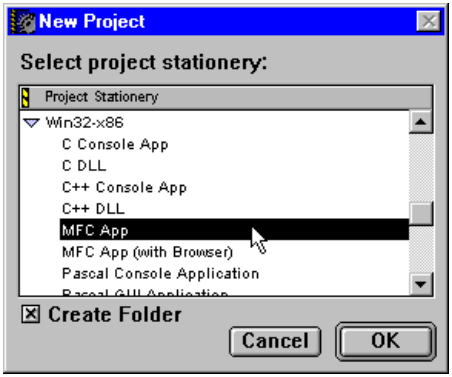
In this exercise you are going to use the MFC to create a simple application so you can see how to use the MFC in a CodeWarrior project. If you already know MFC programming, this exercise gets you on your way quickly. If you don't know how to use the MFC, this exercise shows you how easy it is to open the door to an entire world of programming power—and give you access to a modern, extensible, and complete application framework for Windows programming. (The same general principles apply to both the AWT and PowerPlant as well.)

1. Launch CodeWarrior and create a new project. If CodeWarrior is not already running, launch CodeWarrior, and then choose the New Project item (Shift+Ctrl+N) from the File menu to create a new project. When the New Project dialog appears, do two things:

1. Turn *off* the Create Folder check box
2. Select the MFC Application stationery

Then click the OK button. Figure 10.7 shows you the dialog box.

FIGURE 10.7
Select MFC Application as your stationery, and don't make a new folder.



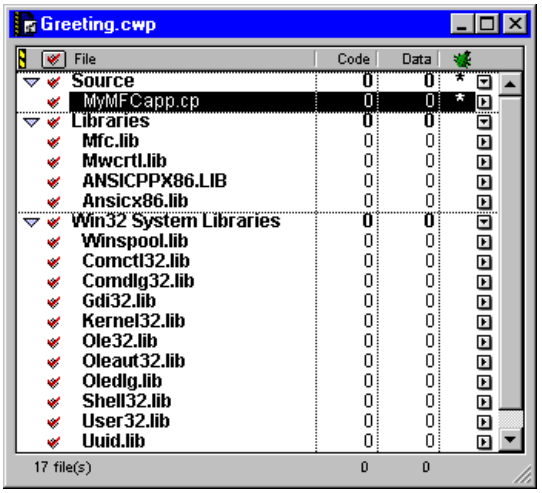
When you click the OK button, the standard dialog box appears to save the new project. Name the project Greeting.cwp, and save it inside the Start Code folder for this chapter, inside the Exer10 folder. When the new project window appears, notice that it includes the Mfc.lib file along with the MSL and several Windows libraries.

2. Replace the main.c file. The project stationery has a default placeholder file named main.c that doesn't do much. In this step you replace that file with a new source file.

First, select the main.c file so the new file you are about to add appears next to it in the same group. Then use the Add Files command

in the Project menu to locate the MyMFCapp.cp file (it's in the Start Code folder) and add it to the project. Finally, select the main.c file in the project window and remove it from the project—use the Remove Files item in the Project menu. When you're finished, the project window should look like Figure 10.8.

FIGURE 10.8
An MFC project after replacing the placeholder file.



3. Examine the header file. In this step you look at the MyMFCapp.h header file. Use the Open command in the File menu to locate and open this file. Notice that it declares two new classes, CMyWindow (which descends from CFrameWind, an MFC class) and CMyApp (which descends from CWinApp, another MFC class). The CMyWindow class has a constructor, and CMyApp overrides the CWinApp **InitInstance()** function. Clearly these classes are using and rely on the MFC.

The particularly astute observer might notice that something is missing. Nowhere is there a **#include** statement that incorporates the MFC header files that declare CFrameWind or CWinApp. Yet this code compiles correctly. Because you used stationery, the project's pre-

fix file includes the MFCHeaders precompiled header. Take a look at the C/C++ Compiler settings, and you'll see it. As a result, you don't need to separately include the MFC header files.

4. Write the CMyApp::InitInstance() function. In this step you write code that calls functions in the MFC. Open MyMFCApp.cp, locate **InitInstance()**, and fill in the body of the function with the code below. I've included existing code in italics to let you know where the new code goes.

```
BOOL CMyApp::InitInstance()
{
    m_pMainWnd = new CMyWindow();                  // create the window
    m_pMainWnd->ShowWindow(m_nCmdShow);  // show it
    m_pMainWnd->UpdateWindow();                    // draw it
    return TRUE;
}
```

This code creates a new CMyWindow object, tells that object to make itself visible, and then draws the window. The calls to **ShowWindow()** and **UpdateWindow()** use functions inherited from CWindFrame, the MFC class from which CMyWindow inherits.

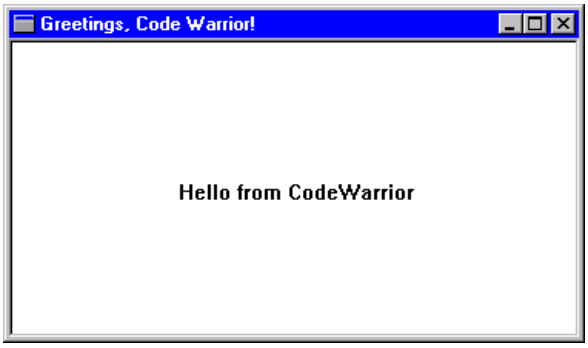
After you enter the new code, save your changes and close the file.

5. Run the project. You're all done. The remaining code (to create an instance of the application, the window's constructor, and code to draw in the window) has been provided for you. Choose Run from the Project menu to compile, link, and launch the application. The entire process takes a few moments, because CodeWarrior has to locate and read the MFCHeaders file, compile the source code, and then add every library to the project.

If you run into errors, revisit the earlier steps to ensure you got the code right. When you run the project successfully, you'll see a new window appear with an intriguing title, as shown in Figure 10.9. Go ahead—resize the window, move it around, and do all sorts of other

window-related things. It all works, and you didn't have to write any of the support code for window behavior. You got it for free from the MFC.

FIGURE 10.9
Greetings, code warrior!



Congratulations! You have created an MFC application from scratch. Clearly, programming with the MFC has a lot more to it than creating a simple window, but no matter how complex your code, the fundamental principles you have just practiced remain unchanged. Using the MFC in a CodeWarrior project really is this easy.

As always, feel free to continue exploring on your own. For example, you might want to run this code under debugger control. You already know all you need to know to accomplish this goal—you are becoming a master of CodeWarrior.

Summary

This chapter introduced you to the concept of a framework, and how it affects the issue of programming for a modern GUI. You know that CodeWarrior provides you with the world-class code you need to program for the target you want, and you know how to get that code into a project where you can use it.

One critical aspect of GUI programming is how you actually create the visual interface that establishes the look and feel of your application—the menus, windows, buttons, icons, and other controls you provide. That's what we'll talk about in the next chapter—Constructor, a graphical interface builder that lets you create a program's interface interactively.

CHAPTER 11

ELEVEN

Using Constructor

- Understanding Constructor
- Managing Resources
- Editing Layouts
- Editing Menus
- Editing Bitmaps
- Editing Text Traits
- Cool Features
- Hands-On Exercises

Constructor is Metrowerks' rapid application development (RAD) tool. It is a fully interactive graphical user interface builder that lets you design a window or a menu much like you would use a drawing program to create a picture. It is simple, intuitive, and very powerful.

NOTE

The CodeWarrior RAD tool is undergoing dramatic changes. With the success of dynamic languages such as Java, new power can be added to the tool. I cannot predict for certain what evolution or path the tool will follow. Constructor as it now stands—a tool to System 7 Mac OS applications from Windows—is likely to change profoundly. Whatever path RAD development takes, it is sure to provide an integrated, multitarget solution.

The Thought behind Constructor

A visual interface is like a picture. It consists of a variety of elements such as windows, controls, menus, icons, bitmaps of various kinds, and so forth. The effort required to write code that creates, locates, and manages these items is significant. A GUI builder is designed to make that process faster and easier for you.

Constructor is a GUI builder. You use Constructor to create a complete interface for an application, including all its menus, pictures, buttons, windows, controls, and so on. Although this is a separate tool, the designers know that you're using this tool as part of a complete development environment. So a lot of care went into the design to make Constructor seem as familiar to you as possible.

continued ➠

The Thought behind Constructor

This program uses the same project metaphor as the IDE, and many of the interface elements in Constructor are the same as in the IDE's project manager. Beyond that, Constructor has ambitious goals. Of course, a full set of standard interface elements is available, but you will also be able to create and save custom interface elements in catalogs or collections, and switch to them quickly and easily. Once you have created a useful little widget for yourself, you'll have it readily available in any project at any time.

For now, you can see Constructor for what it is—an outstanding GUI-builder for Mac OS using PowerPlant. But keep in mind that new tools are coming that help you design a human interface for a Windows, Java, or Mac OS program.

Understanding Constructor

Constructor is a project manager for building a visual interface. As such, it has many of the features of the IDE project manager. Figure 11.1 shows you a simple Constructor window, which lists the kinds of elements you can work with in Constructor. The headings are analogous to group headings in the IDE. Each of the headings is a hierarchical entry that you can open or close with the familiar disclosure triangle.

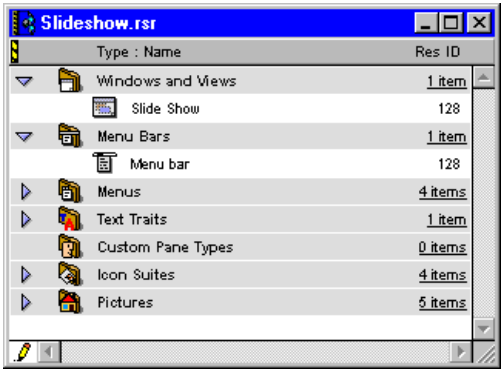
Unlike the IDE, in Constructor you cannot define your own groups. You have available to you a group for each kind of element Constructor understands. They are:

- **Windows and views**—a window and its contents (I'll call these *layouts* to avoid terminal terminology confusion when discussing such things as windows for editing windows)
- **Menu bars and menus**—an application's menus

- **Bitmaps**—icons, pictures, and so forth
- **Text traits**—a description of text appearance
- **Custom panes**—custom interface elements

FIGURE 11.1

Constructor uses a project metaphor, just like the CodeWarrior IDE.



Remember that Constructor is (at this time) designed for Mac OS. On Mac OS, each of these different kinds of elements is represented by a resource. A *resource* is a chunk of data in a standard format that can be understood by any code that knows the standard. A particular application can reconstruct any item based on the information stored in the corresponding resource. So, when targeted at Mac OS, Constructor is fundamentally a resource editor.

NOTE

To understand Constructor and use it effectively, you must have some understanding of the target framework. For Mac OS, that means you should be familiar with PowerPlant. Metrowerks has excellent documentation for this framework and for Constructor. *The PowerPlant Book* covers the fundamentals, *PowerPlant Advanced Topics* discusses more intricate issues, and *Constructor Manual* provides you with all the details about Constructor.

The Mac-hosted set of CodeWarrior tools already includes Constructor for Java, and the Java-targeted Constructor generates some source code to recreate the interface at runtime. The mechanism for storing the interface data for the Windows target is undetermined and depends on the final shape of the targetable RAD tool in the works.

Working with Constructor

Constructor is designed to be as intuitive and simple to use as possible. Toward that end, in many ways it looks and acts just like the IDE. The friendly disclosure triangle has already made an appearance in Figure 11.1. Where the IDE project manager gathers up a collection of files, Constructor collects a series of individual resources.

You can work with an individual resource or group of resources as a whole—just like you can work with a single file or group of files in the IDE. You can create, move, copy, and delete a resource or group of selected resources. You can also work with the contents of a resource—editing a particular menu for example—the same way you can work with the contents of a source file in the IDE.

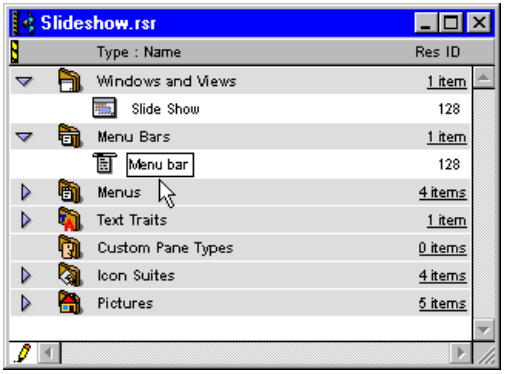
Constructor follows all the usual conventions in its own interface design. For example, you select an item or group of items first, and then you act on it. You select items in all the usual ways: you can click an item, use the arrow keys to navigate through a list, extend a selection using the Shift key, or click and drag the mouse over a group of items to select them all, and so on.

Constructor uses drag-and-drop extensively to enhance the feeling (and reality) that you can directly manipulate the data you're working with. So, if you want to move an item—an entire resource, a menu, or a button—from one place to another, you just drag it to where you want it to be and drop it there. This approach extends across all of Constructor. You can drag items within a window, and between windows. Drag-and-drop also works with multiple selected items as well, not just one item at a time.

To copy an item, you can use the traditional commands in the Edit menu, but you can also Alt+drag the item—you move the item to a new location, but leave the original in place, thus creating a new copy.

Many items have text elements—the name of a resource, the text in a menu, and so on. To edit a text item, simply click it to select it. Then move the cursor off the item, or just wait a second. The text item becomes an editable text field, as shown in Figure 11.2. If multiple text fields are available for a single item, you can use the Tab key to move among them in one direction and Shift+Tab to go in the other direction, just as you would expect.

FIGURE 11.2
Editing text in place is another friendly Constructor feature.



Constructor also supports single-level undo, which means that you can undo and redo the most recent action.

In summary, the Constructor interface is exactly what you would expect from a good interactive program, with the additional cool feature that almost everything you want to do you can accomplish using the direct manipulation techniques of drag-and-drop. This feature lets you interact so closely with your data that sometimes Constructor becomes completely transparent—you just might forget that you're using a design tool.

Managing Resources

At a high level, when you work in Constructor you're working with whole resources. These are the items you see listed in the Constructor project window—the windows and views, menus, bitmaps, and so forth. At this level you create, modify, copy, and delete whole resources. Later on we'll talk about how to open up an individual resource to edit its contents.

You can create a new project with the New Project File command in the File menu. When you save the file, use the .rsr naming extension as a convention for identifying Constructor project files. You open and close existing projects using standard techniques.

Once you have a project file, you can start creating resources. All resources are stored in the same project file, but you can work with individual resources within that file.

Making a New Resource

Constructor has several techniques you can use to create a new resource. The simplest technique is to choose the New Resource item (Ctrl+K) from the Edit menu. Because Constructor works with various resources, you have to tell Constructor what kind of resource you want to create. You can do this in two ways.

If you have already selected a single resource (or the header bar that describes an entire category of resources in the project window) then Constructor assumes that's the kind of resource you want to make. Figure 11.3 shows you how to set up the project window to create a new menu—notice that the menu header is selected.

If you don't have a resource selected, then Constructor displays a dialog box that asks you to specify the kind of resource you want. Figure 11.4 shows you this dialog box. You choose the resource type from the pop-up menu that lists all possibilities.

FIGURE 11.3

Selecting the menu category first tells Constructor you want a menu resource when you choose New Resource.

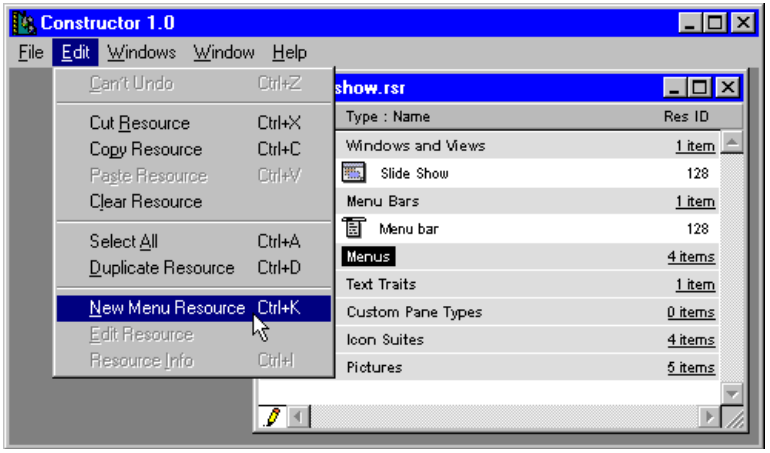
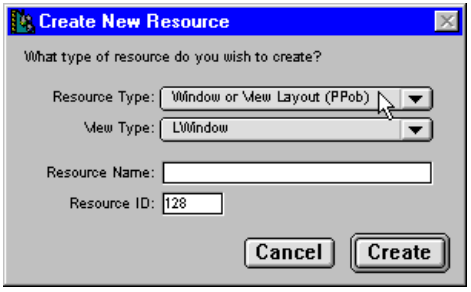


FIGURE 11.4

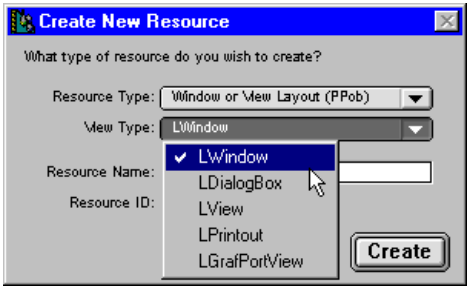
If Constructor cannot figure out for itself what kind of resource you want, it asks you to specify.



If you want to create a new window layout, Constructor always displays this dialog box because you must not only specify the kind of resource, for layouts you must also specify the kind of object you want to use as the top-level container that will hold all the contents for this view. The View Type pop-up menu shown in Figure 11.5 gives you your choices—each of these items represents a PowerPlant view class.

FIGURE 11.5

Set the top-level view in the layout hierarchy using the View Type pop-up in the Create New Resource dialog box.

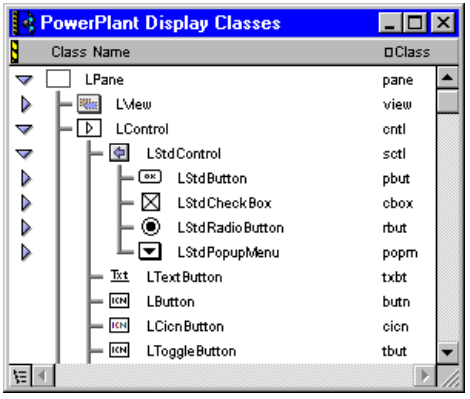


That's pretty simple—you tell Constructor what kind of resource you want either before or while you create a new resource. You can create new resources in other ways as well. For example, you can copy an existing resource as we'll discuss in just a bit.

To create a new layout you have one more surprise technique as well. Constructor displays all available visual objects (the interface elements you can add to a layout) in the Display Classes window—these are the classes that have a visual display. Figure 11.6 shows you this window that is, effectively, a catalog of all available visual objects. If you want to use any of these items as a top-level view for a layout, simply drag it onto the project window and drop it there.

FIGURE 11.6

The Display Classes window catalogs every available visual interface item.



TIP

By the way, note the little button at the bottom left of the window, to the left of the horizontal scroll bar. Look familiar? This is the hierarchy/alphabetical button that switches the view from one style to the other, just as in the class browser.

Because you can use the Display Classes window as the source for a top-level view in a layout, you can create a custom window class (which would appear in the Display Classes window) and use it as the basis for any of your windows. This is a very useful feature.

WARNING

Some items are not appropriate for a top-level view, so you should be careful what you drag over to the project window. Only objects that descend from the PowerPlant LView class can contain other panes, and even some of them are not intended as top-level containers.

While you are creating a new resource, you can also specify the name and ID number. The name is always optional, but is very helpful and a lot more informative than a number. The ID is mandatory, but Constructor always provides a default value that you can accept if you wish.

NOTE

The ID number is used to locate the correct resource at runtime.

Working with Whole Resources

When you create a new resource, Constructor fills it with default information. The odds that this default resource will be useful without further modification are about nil. We'll discuss how to go about setting all the information in each different kind of resource in just a bit.

Before we do, however, let's continue to explore working with resources as a whole.

This process is directly analogous to working with files in the IDE's project window. Now that you have created a resource, you can open, copy, modify, and delete a resource.

You have three ways you can open a resource from the Constructor project window, as follows:

1. Double-click the resource
2. Select one or more resources and press Enter
3. Select one or more resources and choose the Edit Resource item from the Edit menu

You also have several ways to copy or duplicate a resource. You can copy and paste a resource, or choose the Duplicate Resource (Ctrl+D) item in the Edit menu. You can also just drag a resource from one project window into another—this does not remove the original from the source project.

NOTE

You cannot Alt+drag to copy a resource in the same project window. Use the Duplicate Resource command for that.

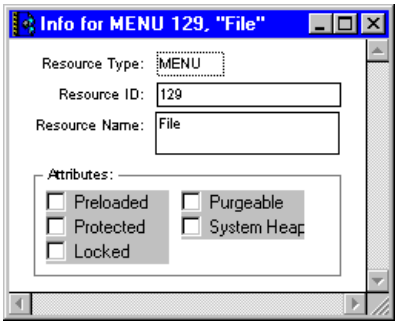
Deleting a resource is as easy as selecting it and pressing the Backspace key. You can also use the Cut command (Ctrl+X) from the Edit menu.

Every resource also has certain characteristics—the name, ID number, and five attributes that relate to how the Mac OS Resource Manager handles the resource in memory. You can edit the name and ID number right in the project window, as shown back in Figure 11.2. Simply put the text into edit mode (select it and wait a second) and then enter the new text.

To modify the memory attributes of a resource, choose the Resource Info item (Ctrl+I) in the Edit menu to open the Info dialog box shown in Figure 11.7. You can modify the name and ID number here as well, and set the attributes using the check boxes. Both PowerPlant and Mac OS work correctly if all attributes are off, and that is the default condition for all resources.

FIGURE 11.7

You can use the Info dialog box to modify the memory attributes of a resource, but think twice before you change them.



WARNING

Don't modify the memory attributes unless you are thoroughly familiar with the side effects. For example, marking a resource as purgeable means that it can be removed from memory by the operating system without notice to your application. This can cause disaster if you don't allow for the possible absence of the resource.

Revisiting the Resource Fork

You may recall from earlier in this book that certain information required for a CodeWarrior project is stored in a shadow file in the Resource.frk folder that exists alongside the master project. You'll also recall that this doesn't cause any problems unless you start moving files around manually on the Windows desktop. Constructor uses the same Resource.frk strategy, so let's take a look at this one more time because it is a significant issue, particularly when you want to transfer your Constructor files from one platform to the other.

Constructor is a resource editor. For all intents and purposes, *all* of the information you create when working in Constructor ends up in the shadow file. If you examine the properties of the master .rsr file (the one that is not in the Resource.frk folder), you'll discover that it occupies zero bytes! Don't be alarmed. You really are saving data, it's just all in the shadow file in the Resource.frk folder.

The master file is still very important. You never work directly with the shadow file in the Resource.frk folder. For example, when you open a file you always open the master file. The only time you ever have to deal with a shadow file is when you want to manually move a Constructor .rsr file on the Windows desktop. At that time, remember to move both parts of the bifurcated file and maintain the same hierarchical relationship between the master file and the shadow file.

When you want to move your file over to a Mac OS computer, move both parts onto a removable disk, and make sure they stay in the same relationship to each other—that is, the shadow file stays in a directory named Resource.frk that sits in the same directory as the master .rsr file. Then move the disk over to a Mac OS computer that has the Apple-supplied utility PC Exchange installed. Copy the file onto your Mac drive, or use Constructor on the Mac to open the master file from the disk, and then save it on your Mac. The Resource.frk folder is invisible on the Mac OS desktop, so you cannot see or open the shadow file, but PC Exchange puts the two parts back together again automatically.

Moving a Constructor project file from Mac OS over to Windows is a bit easier. Simply copy the file onto a DOS-formatted removable disk on the Mac OS computer. Then move the disk to your Windows machine. Once you arrive, make sure the file has the .rsr naming extension, and place this file inside a Resource.frk folder—that's right, this is the shadow file. Then simply create a dummy file to use as the master file, place it outside the Resource.frk folder, and give it the same name as the shadow file.

That covers how you work with files and whole resources with Constructor. It's very straightforward, and very much like the IDE. Now let's see how to edit the content in different kinds of resources. The principal resource types are

- Layouts
- Menus
- Bitmaps
- Text traits

Each of these resources is a unique kind of data, and editing the contents of each resource presents unique issues.

Editing Layouts

For layouts, Constructor targeted at Mac OS depends upon the PowerPlant application framework. In this context, a layout is not only the window frame, but everything in that frame including scroll bars, buttons, edit fields, pop-up menus, and so on. PowerPlant uses a special resource called a PPob (which stands for PowerPlant object, and is pronounced pea-pob) to store the information it needs to recreate a complete layout. All the information about a layout—including its appearance, size, location, contents, and the containment relationships of its contents (that is, which objects are inside which other objects)—is stored in the PPob resource. At runtime, PowerPlant's universal object-reanimator code reads the appropriate PPob resource data, and reconstructs the entire layout (including the encompassing window) based on the information it finds. Writing any application-specific code is not necessary at all, except the one line of code to start the layout-creation process and any special initialization code required for custom interface elements. All the other information about every object in the window is created, managed, stored, and edited at design-time by Constructor.

Using Layouts and Hierarchies

Data in a layout is organized in two very different ways. One relationship is spatial—you have all these items in the layout, and they have a spatial relationship one to the other. For example, a caption appears next to a picture, and both are beside a scroll bar. When you look at this information, you're really looking at the visual appearance of the window. In Constructor, you look at this information in a layout view.

However, a very different, invisible, but nonetheless critical relationship occurs among the objects in a layout: containment. In many cases, you have visual items that are literally inside other items, not just overlapping. For example, you may have a large picture inside a smaller viewing area that scrolls. In this case, not only is the picture item spatially beside the scroll bar, it is also hierarchically inside a scrolling view.

When you open up a layout resource, by default Constructor shows you the layout view. Figure 11.8 shows you a typical layout view. Notice how each object has an outline, and an ID number. This helps you keep track of what's what, and where everything is. You can turn off the outline and the ID number if you wish, using commands in the Layout menu.Looking at a layout, you can tell where objects are with respect to each other. But if two objects overlap, is one of them inside the other or is it just on top of the other? This is a containment issue, and to answer the question you really need to see the data from a different perspective. You can look at the same data from a hierarchical perspective, and Figure 11.9 shows you what that looks like. Objects contained within other objects are inset, and objects that are containers have a disclosure triangle. Look familiar?

Containment is a critical issue in how a layout works. For example, if the container is a scrolling view, an object inside it can scroll around, but an object on top of it will not. Both views—layout and hierarchy—are

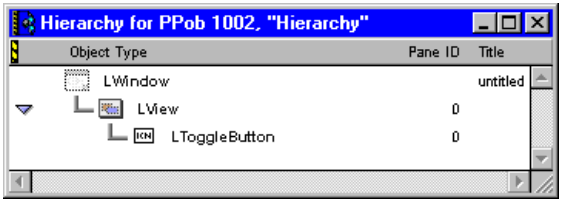
really separate windows on the same data. So if you select an object in one view, it becomes selected in the other. If you delete the object in one, it disappears from the other. The two views remain consistent at all times, because they each represent the same underlying resource.

FIGURE 11.8

The layout view lets you see the spatial relationships among a layout's contents.

FIGURE 11.9

The hierarchy view shows you which objects are inside which other objects in a layout.



TIP

By the way, if you want to click a pane to select it, but can't because it is in a stack of overlapping panes, you can Ctrl+click repeatedly to cycle through each pane in the stack until you get the one you want.

Adding a Pane to a Layout

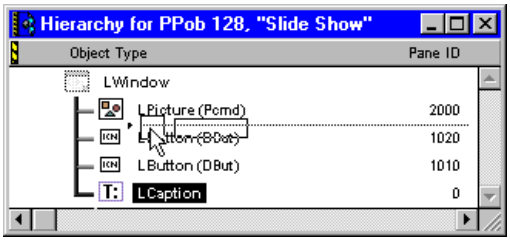
When you create a new layout (which is the visual representation of a PPob resource), it is empty. The first thing you want to do is add some objects to the layout. I'll use the term *pane* to represent any such object. A pane might be a check box, a text field, a scrolling view, a picture, or any of dozens of possible objects.

To add a pane to a layout, drag it from the Display Classes window and drop it in the view. You can directly manipulate the size and location of the pane. Drag it to where you want it to be, or drag a selection handle to resize it.

TIP You can drop a pane into the hierarchy view as well.

In the hierarchy view, you can adjust the containment relationship just as easily. Select the pane you want to relocate hierarchically, and then drag it to where you want it to go. Figure 11.10 shows you how putting a caption inside a picture works. It can be a little tricky at first figuring out where to drop it to ensure it becomes contained by some other pane, but after you've done this a couple of times it becomes second nature.

FIGURE 11.10
You rearrange the containment hierarchy by dragging a pane to a new location in the hierarchy.



To remove a pane, select any pane or panes in either the layout or hierarchy views, and then press the Backspace key or choose Cut (Ctrl+X) or Clear from the Edit menu.

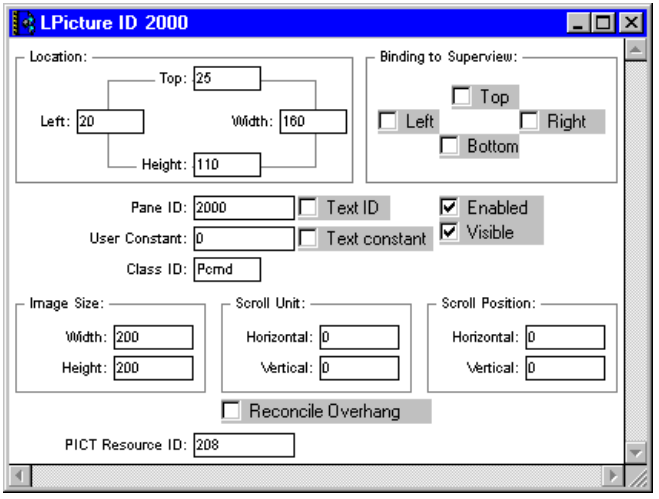
Setting Pane Properties

Panes have many other properties besides size and location. For example, a label has text content, and a connection to a Text Traits resource that defines the appearance of this bit of text. You cannot set these properties directly in the layout window. You must open a property inspector window to access each pane's unique set of characteristics.

To open the window, you double-click the pane. Figure 11.11 shows a typical property inspector window. Each different kind of pane has a different set of attributes, so each inspector is a little different. The *Constructor Manual* that is part of the CodeWarrior documentation has a reference section detailing each property inspector, and specifying what each different property is for.

FIGURE 11.11

The property inspector window lets you modify any attribute of any pane.



You can modify size and location here along with a lot of other traits. You can set the pane's ID number, the class ID—a value that PowerPlant uses to reconstruct the object at runtime—and many more items.

TIP

When a pane contains a picture, and the resource for that picture is inside the same .rsr file as the layout, then Constructor displays the actual picture inside the pane as you edit. This is a great way to get real visual feedback for what your layout will look like. Refer back to Figure 11.8 to see a pane that contains a picture.

Moving and Arranging Panes

The fastest way to move a pane is to simply drag it to where you want it to be. Dropping it at the precisely correct pixel can be difficult, however, because the mouse doesn't always give you the fine control you need to nudge a pane just a pixel or two in any direction. You can move a pane pixel-by-pixel with the keyboard—simply select a pane and then use the arrow keys to nudge it. Every time you press a key, the pane moves one pixel in the corresponding direction. You can also open up the property window for a pane and enter the actual coordinates you want that pane to have in the layout.

While you can certainly arrange your panes one at a time, as a practical matter you don't want to tweak a series of individual panes just so they all line up nicely along the left edge. Constructor has features that give you tremendous flexibility with respect to arranging panes aesthetically.

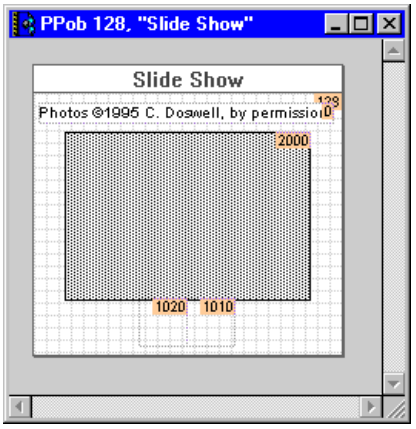
Using the Grid

At a very simple level, you can display a grid on which you place panes. You can use the grid as a rough-and-ready alignment tool—just move the pane until it hits the grid line you want. You can show or hide the grid using the Show/Hide Grid command in the Layout

menu. The text in this menu item changes depending upon whether the grid is currently visible or not. Figure 11.12 shows a layout view with the grid displayed.

FIGURE 11.12

You can display an alignment grid to assist you in arranging panes.



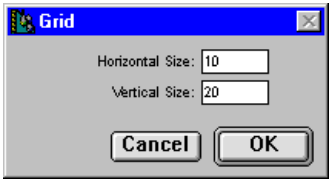
You can snap items to the grid (or not) by choosing the Snap/Don't Snap to Grid item in the Layout menu. The menu item changes text, and as with the Show/Hide Grid item, the actual state of the application is the opposite of the text in the menu item. The menu item tells you what will happen if you choose the item. So, if Snap to Grid is the menu item text, that means snapping is currently off, but if you choose the item all future drags will snap to the grid. When this feature is on and you drag a pane to move it, the pane automatically moves to the nearest grid lines—both horizontal and vertical.

TIP

If you press the Ctrl key *after* you begin dragging, the snap to grid state is reversed. For example, if this feature is on, then pressing Ctrl while dragging turns it off temporarily. If this feature is off, pressing Ctrl turns it on as long as the key is pressed.

You can modify the size of the grid as well. Choose the Edit Grid item (Ctrl+E) in the layout menu to display the dialog box shown in Figure 11.13. Note that the grid does not have to be square—you can set different spacing for the horizontal and vertical directions.

FIGURE 11.13
The Grid dialog box lets you determine the spacing of lines in the grid, in pixels.

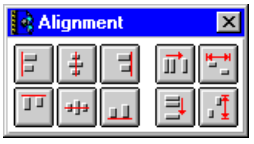


Aligning Panes

While the grid is very helpful, it still requires a pane-by-pane tweak. Constructor also has the ability to align several selected panes automatically. As is so often the case with CodeWarrior tools, you can take more than one path to do this task.

The easiest and most intuitive path is to use the alignment palette shown in Figure 11.14. You display this palette by choosing the Alignment Palette item in the Windows menu. The buttons on this palette let you align selected panes along the left edge, horizontal center, right edge, top edge, vertical center, or bottom edge of the selected panes.

FIGURE 11.14
The alignment palette lets you arrange selected panes automatically.



The other buttons on this palette, along the right side of the window, are used for distributing selected panes within a given space. We'll talk about that in just a bit. Right now, let's look at an alternative mechanism for aligning panes.

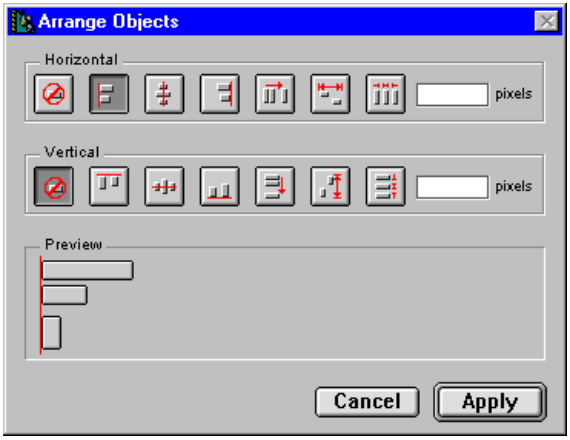
TIP

Commands are also available in the Arrange menu for each alignment option.

Rather than use the alignment palette, you can display an alignment dialog box. Choose the Arrange Objects item (Ctrl+L) in the Arrange menu to display the dialog box shown in Figure 11.15. You'll see the same alignment buttons, and they do the same thing.

FIGURE 11.15

The Arrange Objects dialog box duplicates and enhances the features of the alignment palette.



You might prefer to use the dialog box instead of the tool palette. And for good reason—because you can see the effect of multiple commands, such as top left, without committing yourself. The dialog box lets you preview the change. The actual alignment change in your

layout doesn't take place until you click Apply. This lets you experiment with different alignments to see how you like them. By contrast, clicking a tool palette button causes an instantaneous change in the actual layout. Both techniques are equally sound—one is fast, the other safe. Use either or both as you see fit.

TIP

Remember, you can undo your most recent action—but only the most recent action. So if you're using the alignment palette and make a mistake, undo it right away.

Altogether, Constructor gives you three ways to align objects automatically—menu choice, tool palette, and dialog box. You have the same kind of flexibility when it comes to distributing panes as well.

Distributing Panes

Rather than lining up a series of panes along an edge, you might want to space them out within a given area. Distributing panes works just like aligning—you have menu commands, buttons on the tool palette, and buttons in the Arrange Objects dialog box. Figures 11.14 and 11.15 show you the distribution buttons. You should note that the dialog box in Figure 11.15 has some buttons that aren't available in the alignment palette.

Table 11.1 lists each distribution option, the corresponding icons, and the range within which you are spacing out the panes. Icons for both horizontal and vertical distribution are available for each option.

If you choose to spread panes within a container, the container is the pane that contains (hierarchically not visually) all the selected panes you are distributing. If the panes all appear within a single scrolling view, for example, then the bounds of that view control the spread.

TABLE 11.1 Distribution options and corresponding icons and ranges

Option	Icons	Range
Spread		Between outermost selected panes
Spread in container		From edge to edge of the container that holds all the selected panes
Spread by gap		From the top or left selected pane, with the specified gap between panes

If you want to spread the panes with a specified gap between them, you provide the values in the Arrange Objects dialog box. Neither the alignment palette nor the Arrange menu support this option.

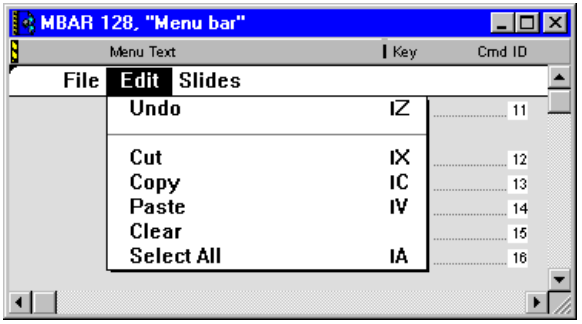
This brief survey of the features you use to edit a layout—either visually or hierarchically—give you a real flavor for how intuitive and direct Constructor's approach is to resource editing. Except for pane properties, just about everything you want to do you can accomplish with drag-and-drop functionality. It is really like working at your desk—you just pick up the things you want to move, and move them. You can nudge things, line them up, spread them out, drop them inside each other, and so on. The same flexible style holds true for menus.

Editing Menus

Editing menu resources with Constructor is a snap. Essentially, you have two sorts of resources to consider—a menu bar and a menu. On Mac OS, the menu bar is a collection of the menus that appear across the top of the desktop when the application is in the foreground. Because the menu bar encompasses most of a typical application's menus, editing from the menu bar is the most efficient way to work with menu resources in Constructor. You pick the menu you want to work on, then edit its contents. Figure 11.16 shows you the menu bar editor in action.

FIGURE 11.16

The menu bar editor shows you all the principal menus in the application, and lets you open them for editing.



The resource that stores information about what menus appear in the menu bar, and what order they appear in, is called an *MBAR resource*.

A project may also have menus that do not appear on the menu bar, such as submenus, or pop-up menus that appear at various times. Although you cannot access these menus through the menu bar, you can use the individual menu resource editor, which has all the same features.

Each menu item has a name and a unique command ID. The command ID is the message the application receives when that particular menu item is chosen by the user. A menu item may also have a keyboard shortcut associated with it. In Mac OS terminology, this is called a command key, because you press the Command modifier (analogous to the Ctrl key on Windows machines) while typing the shortcut key to issue the menu command.

TIP

For a full discussion about how PowerPlant uses menu command IDs and manages menus, refer to *The PowerPlant Book* that is part of the CodeWarrior documentation, particularly the chapter on commanders and menus.

In this section we'll talk about how you create and add menus to the menu bar, and how you create and edit items in a menu.

Making a New Menu

To begin, create or open a menu bar resource. When the menu bar editor is open, you can choose the New Menu item (Ctrl+M) on the Edit menu, and you have a new menu.

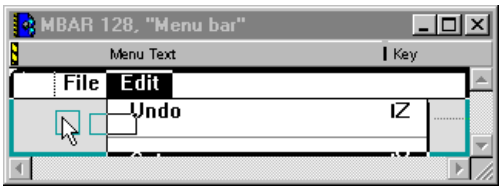
On Mac OS, the first three menus must be the Apple menu (a color Apple icon), which is analogous to the Windows Start menu, the File menu, and the Edit menu. If the menu you are creating is one of the first three in the menu bar, Constructor automatically gives it the correct title. Otherwise, the menu is named "untitled" by default, but you can change the name.

You can also create a new menu resource using the project window, as outlined earlier in this chapter. You can then add that existing resource to the menu bar. Let's discuss how to manipulate the contents of the menu bar, and then talk about individual menus.

Working with the Menu Bar

Let's say you have a project that already has a series of menus, and you want to add this menu to your menu bar. You can simply drag the menu resource from the project window and drop it in the menu bar editor. Figure 11.17 shows this process in action. As you move across the menu bar window, the current insertion point for the new menu flashes so you know where you're dropping it.

FIGURE 11.17
You can add an existing menu to the menu bar with drag-and-drop.



You can move menus around in the menu bar using the same technique. Simply drag the menu title around in the menu bar until it is where you want it to be, and then drop it there. It's as easy as shuffling papers on your desktop. You can also cut and paste the menu, but dragging is so much simpler. If you paste a menu, it appears after the currently selected menu, or at the end of the menu bar if you have nothing selected.

Removing a menu from the menu bar is equally simple. Select it and press the Backspace key, or use the Cut (Ctrl+X) or Clear items from the Edit menu. Whatever technique you use, the menu disappears from the menu bar. This operation does *not* delete the corresponding menu resource from the project.

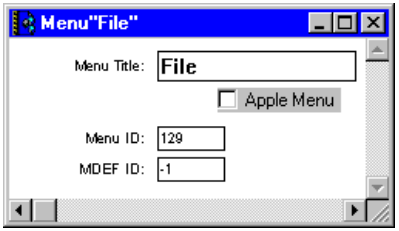
TIP

You can select a menu or a menu item by clicking the text or using the arrow keys. The left and right arrow keys change menus. The up and down arrow keys change menu items. You can select multiple menu items, but only one menu at a time.

To change the title of a menu, select the menu, put the text in edit mode, and type away. You can also select the menu title and choose the Menu Info item (Ctrl+I) from the Edit menu. Doing so displays a little dialog box, shown in Figure 11.18, in which you can set the menu title and also modify the menu ID and MDEF ID if you feel like living dangerously.

FIGURE 11.18

The Menu Info dialog box lets you set various properties of a menu.



WARNING

Don't mess with the menu ID. This is not the same thing as the menu *resource* ID, but most Mac OS programmers this side of the moon treat them as if they were the same. Constructor automatically sets the menu ID to the same value as the menu resource ID, and thus all is right with the world.

Editing an Individual Menu

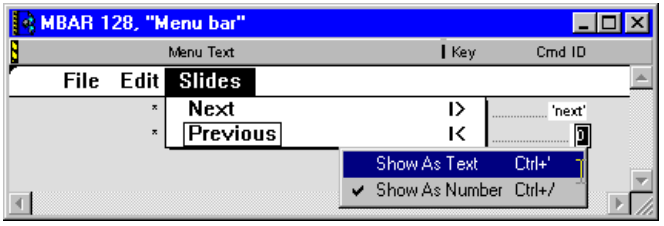
After you have a new menu, you must put some items on the menu. Use the New Menu Item command (Ctrl+K) in the Edit menu, and a new untitled item appears in the menu. If you want a separator bar, use the New Separator Item command (Ctrl+hyphen).

When you create a new item, it appears ready to have its content edited. Just type in the text you want to use for the item. You can use the Tab key to jump to the command key, and then to the command ID and set those values as well.

Menu command IDs deserve a special note. Although a command ID often has a numerical value, it can also be a text value. Here's how to go about using a text ID. First, put a command ID item into edit mode, and then Alt+click on the item to see the pop-up menu illustrated in Figure 11.19. This menu lets you choose how you're going to use this command ID—as a number or as text. If you choose text, you can enter a string of four characters. If you use digits at this time, they are interpreted as text characters, not as numbers.

FIGURE 11.19

Alt+click the command ID to choose number or text entry for this item.



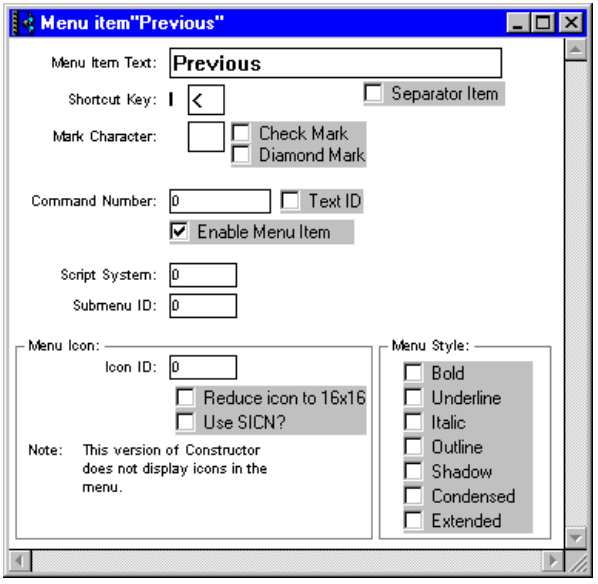
You move a menu item exactly the same way you move a menu. Just drag it where you want it to be. This procedure even works across menus, which is very cool. You can drag a menu item to the menu bar, slide down the bar opening up menus as you go until you get to the one you want, then slide down the destination menu and drop the item in place. The command key and command ID move right along with it.

Menu items have properties as well, and of course you can set each and every one of them in Constructor. Just double-click the item, or select it and choose the Menu Item Info (Ctrl+I) command in the Edit menu. Figure 11.20 shows you a typical item's property window. As you can see, you can edit a variety of aspects of the particular menu item, including its name, command key, command ID, text style, whether it uses an icon, and so on.

To delete a menu item, select it and press the Backspace key.

FIGURE 11.20

The property inspector for a menu item lets you set all the attributes for that item.



NOTE

Some menus cannot be fully defined until runtime—the font menu is a perfect example. You create the font menu and add it to the menu bar, but you typically have no items in the menu because you have no idea what fonts will be available. See *The PowerPlant Book* for details on how PowerPlant handles this kind of menu, and how it creates command IDs on the fly called *synthetic* command IDs.

Adding a Submenu

Some menu items are really a heading for a submenu. Creating such an item in Constructor involves going to the menu item's property window. When you open the property window, you'll see that one of the fields is the Submenu ID. By default this value is zero, which means no submenu exists.

If you want a menu item to be hierarchical, that is, to have a submenu appear when you click the item, set this value to a number from 1 to 255. That number represents the menu ID for another menu that appears as the submenu. You must also have a menu resource with that ID available in the application for the submenu to appear, but you can use Constructor to build the menu quickly and easily. The actual submenu does not appear in the Constructor menu bar editor, but you can open and edit each submenu individually with the menu editor.

Once again you can see that Constructor gives you direct access to virtually every aspect of menu editing. You hardly ever have to open a dialog box or enter data in a field or click an OK or Apply button. The drag-and-drop interface gives you quick and intuitive control over the location of menu items, and even lets you move items from one menu to another.

You can see why Constructor—a programmer's tool—has been hailed as a worthy model for application interface design.

Editing Bitmaps

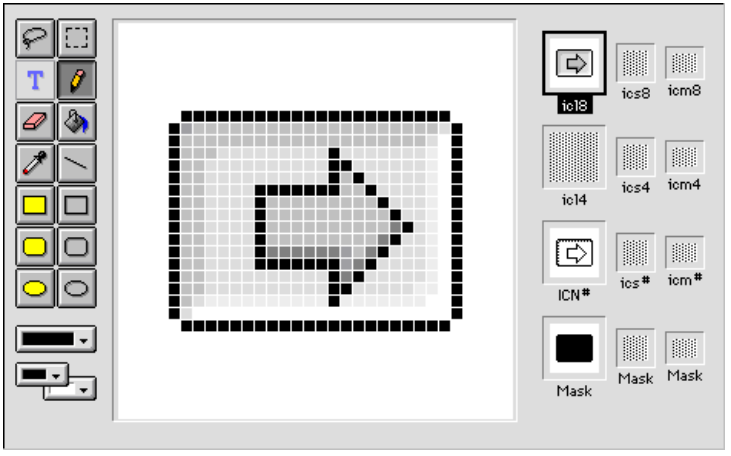
Several resources incorporate bitmaps. To edit a bitmap you need a drawing tool. The Constructor bitmap editor is really quite a capable drawing program in its own right. The same bitmap editor is useful for all bitmap resources. For example, you use the bitmap editor to work with pictures, icons of all kinds, background and fill patterns, and cursors. Whenever you open up one of these many different kinds of resources to edit its content, the same bitmap editor appears that is shown in Figure 11.21. Some of the editor's features are particular to certain kinds of bitmap resources, and we'll talk about them as we go along.

NOTE

At the time of this writing, the bitmap editor was still under development. Some features described in this section may function differently, or not at all.

FIGURE 11.21

The bitmap editor is useful for any resource that contains a bitmap.



Understanding the Bitmap Editor

As you can see, a tool palette appears along the left side of the bitmap editor window. The editing view in the middle is where you do your drawing. Along the right are actual-size sample views for the resource you're working on.

NOTE In some cases no sample view appears. This happens when the resource you're working on in the editing view is already at its actual size, as with pictures.

Table 11.2 describes each tool.

TABLE 11.2 Bitmap editor tools and their purposes

Tool	Icon	Purpose
Lasso		Select irregular areas
Marquee		Select rectangular areas
Text		Enter text
Pencil		Draw pixel by pixel, usually in the foreground color
Eraser		Erase pixels with the background color
Paint		Fill an area of contiguous pixels with the current pattern
Dropper		Set foreground or background color
Line		Draw a line with the foreground color
Filled rectangle		Draw a rectangle filled with the current pattern
Empty rectangle		Draw a rectangle outlined with the foreground color
Filled rounded rectangle		Draw a rounded rectangle filled with the current pattern
Empty rounded rectangle		Draw a rounded rectangle outlined with the foreground color

TABLE 11.2 Bitmap editor tools and their purposes (continued)

Tool	Icon	Purpose
Filled oval		Draw an oval filled with the current pattern
Empty oval		Draw an oval outlined with the foreground color
Hot spot		Mark the cursor hot spot
Pattern		Set the fill pattern
Color		Set the foreground or background color

Using the Bitmap Tools

Click a tool to make it the active tool (the button will look like it has been pushed). Then apply the tool to the drawing area to draw, fill, erase, select, and so on. Drawing is a pretty intuitive process, and you shouldn't need a whole lot of guidance here. However, you'll be glad to know some neat tricks hiding behind these tools. These sneaky little features are both cool and difficult to discover other than by accident. So let me clue you in on a few hot tips for the Constructor bitmap editor.

You can double-click the lasso tool to select the outermost outline of nonbackground color pixels in a view. If you have an irregular shape sitting on the background, you select it precisely with a simple double-click. Double-click the marquee tool to select the entire bitmap.

If you double-click any pixel with the pencil tool, that pixel turns into the background color. This is a great way to erase a pixel here and there very precisely without having to resort to the eraser. While we're on the topic of erasing, double-clicking the eraser wipes out the entire bitmap, but you can undo the operation if you accidentally destroy your prized artwork.

If you click a pixel with the dropper tool, you set the foreground color to match that pixel. If you Shift+click, you set the background color. You can do the same thing with the keyboard without switching to the dropper tool—Alt+click to set the foreground color, and Alt+Shift+click to set the background color. None of this actually changes the pixel you click, just the foreground or background color for your next action.

You can constrain the motion of any tool (such as the line tool) by holding down the Shift key. Using the Shift key with the line tool keeps the line at a multiple of 45 degrees. Using Shift with the oval tool keeps the shape a circle. And using Shift with a rectangle tool or the marquee selection tool restrains the shape to a square.

Working with Sample Views

The sample views on display in the bitmap editor are more than just a visual aid that show you what the bitmaps look like. They are, in fact, useful controls that you use when working on bitmaps. When you click to select a sample view, that view's bitmap appears in the editor window so you can work on it. The currently selected sample view has a heavy border around it highlighting the view.

For some resources, such as icon families, many sample views are available. Figure 11.21 shows you that an icon family has a dozen samples, and you can select any one of them to edit. When two or more sample views appear in the bitmap editor, you can drag-and-drop bitmaps between them.

For example, suppose you have designed an icon that uses 256 colors (an icl8 resource). You want a comparable icon that uses only 16 colors (an icl4 resource). You don't have to start over from scratch. Just drag the bitmap from the icl8 resource sample view to the icl4 resource sample view and it appears. Constructor automatically converts from 256 colors to the closest 16 colors in the available color

palette. You might want to tweak the result, but even if you do, Constructor has done most of the work for you.

NOTE Going from a resource with fewer colors to one with more colors does not add new colors to the bitmap.

You can use the same trick to go from large to small icons—Constructor automatically scales the result to fit the new size, although such scaling doesn't usually result in a particularly aesthetic result. You can even drag-and-drop between bitmap editor windows. If you open up more than one bitmap resource, you'll have a bitmap editor for each. You can drag the sample view from one resource to another and drop it there. You could drag a small icon to a cursor for example, and get the cursor bitmap for free.

Editing Pixels

Drawing and erasing is only part of the story when it comes to creating pretty pictures. Once you select a group of pixels, you may want to work on them as a unit. For example, you may want to nudge the pixels one way or the other. While you can use drag-and-drop for that operation, you can also use the arrow keys (as when moving panes in a layout view).

You can copy selected pixels by Alt+dragging the selected pixels to a new location, or by using copy and paste techniques. You can also rotate and flip the pixels with commands in the Options menu that appears when a bitmap editor window is active.

And of course, from time to time you'll want to delete pixels. To delete a range of selected pixels, just press the Backspace key and poof, they're gone.

Working with Colors and Patterns

A critical aspect of drawing a bitmap is color selection. We've already discussed how to use the dropper tool or keyboard equivalents to set the foreground and background colors.

A more direct approach is to use the color tool at the bottom of the editor's tool palette. This tool is really a pop-up menu to a color palette. The tool has two parts, one to set the foreground color and one to set the background color. In a stroke of genius, the interface designer clearly indicates which is which—one of the pop-up icons sits in front of the other. This creates an intuitively obvious distinction between foreground and background colors—I doubt you'll ever have a hard time figuring out which is which.

Most drawing is done in the foreground color. Most erasing happens in the background color. Keep this in mind, because you can use the background color to create interesting effects with the eraser. You can turn the eraser into a separate drawing tool, not just something for wiping out mistakes. For example, if you wanted to set an entire bitmap to light blue, perhaps as a background in preparation for further work, you can set the background color to light blue, and then double-click the eraser.

The colors that are actually available to you with the color tool depend upon your choices in the Colors menu, and upon the nature of the resource being edited. Some resource types do not support color, and others only support limited color palettes.

In addition to color, you can set patterns with the pattern tool. The colors in the patterns depend upon the current foreground and background colors, so any time you change them, you change the color of the pattern as well.

The pattern is used to fill the inside of filled shapes you draw with the shape tools. The paint bucket tool also fills an area with the

pattern—not with a color. If you want the paint tool or a shape to be filled with a solid color, pick the appropriate color and set the pattern to solid fill.

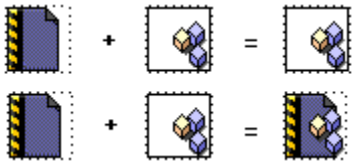
Using Transparency

The bitmap editor has one more great feature, and it really should be classified as a truly cool feature. It is a tremendous help when you're combining bitmaps—for example, pasting a shape onto a background. Here's how it works.

Let's call a group of pixels you copy the *source*, and the bitmap into which you paste this group the *destination*. Assume that your source bitmap has large areas that appear in the background color, but it has an interesting tidbit inside. Without transparency, when you paste the source onto the destination, everything in the destination that is underneath the source bitmap is obliterated and replaced by the source bitmap. That's unfortunate, because what you really want is just that interesting tidbit, not all the white space around it.

When you choose Make Transparent from the Options menu, and then paste the source bitmap into the destination, background-color pixels (usually white, but it could be any color) are ignored. The destination pixels still shine through, effectively rendering all background areas of the source bitmap transparent. Figure 11.22 shows you the effect of no transparency at the top, and using transparency at the bottom.

FIGURE 11.22
Transparency lets you paste a bitmap on top of another without obliterating the destination.



The transparency isn't limited to the "outside" rim of the source bitmap. If interior portions are in the background color, they become transparent as well. This lets you create some neat special effects such as pasting a latticework over a bitmap, or combining bitmaps in other intriguing ways. This capability works with any background color, not just white. In effect, you can use the transparency feature as a filter to remove all pixels of a single color from the bitmap you're pasting. That's really cool!

And that pretty much sums up the beauty of the bitmap editor. Like the layout and menu editors, its simplicity and the ability to manipulate data directly without resorting to dialog boxes or data entry forms gives you a real intuitive feel for working with the pixels.

Editing Text Traits

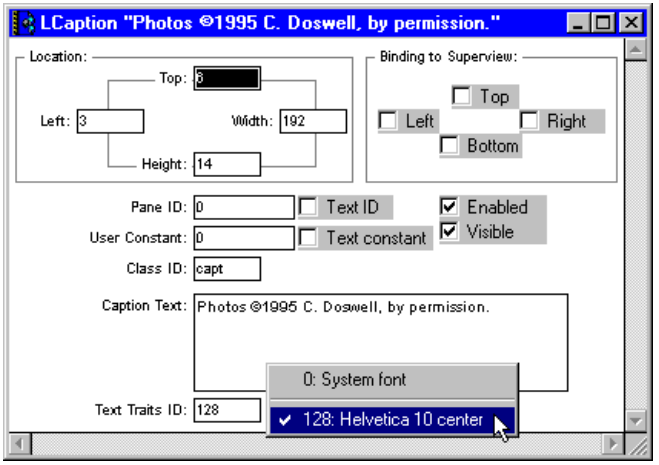
A text trait resource describes how text should appear. You select a font, size, color, style, justification and drawing mode. The resource stores your choices, and you can then use this text traits resource as a handy shortcut to control any particular text-based object.

For example, let's say you have a caption object and you want the text to appear in 12-point Courier bold, flush right, in red. You set up a text traits resource to encapsulate these values. Let's say you give it a resource ID of 128. Then, when you set the properties for the caption, you'll see that it has a text traits field. You can click on the pop-up menu, select the traits you want to apply for this object, and you're done. You don't have to specify the font, size, color, or other attributes directly for the object, You just hook it up to the text traits resource. Figure 11.23 shows you how this works.

Best of all, you can use the same resource for any number of panes. This makes setting up the textual appearance of your program very simple. You just create the text traits resources you need to describe each different kind of text you use, and then set your text-display panes to use them. You might have a hundred panes and three text traits

FIGURE 11.23

The text traits pop-up in a text-display pane's property inspector lets you pick from all available resources in the project.



resources. It's a lot easier to modify those three resources than it would be to specify the text display details for one hundred separate panes.

To create a text traits pane, follow the instructions for creating any resource. To edit the resource, double-click it to open the text traits editor, and just pick the settings you want.

TIP

When you name the text traits resource, give it a name that reflects the settings you pick, such as Courier red 12. This is a lot more informative in the text-traits pop-up menu than the resource ID number.

Cool Features

Constructor is a very cool tool in all respects, but we've got one more neat feature to discuss: custom panes.

The principle is simple—you should not be and are not limited to the default PowerPlant classes. You are able to build your own classes, design property windows for objects of that class, have the class

appear in the Display Classes window (both alphabetically and in its proper place in the PowerPlant class hierarchy), and add objects of your custom class to layouts just as you can an ordinary PowerPlant class. Once your custom class is in the layout, you work with it in exactly the same way you would a PowerPlant class.

In other words, Constructor is fully extensible. You can create and store catalogs of custom objects, and make them available to Constructor automatically so they appear in the Display Classes window for every project. This is *very* cool. Nothing whatsoever distinguishes a default PowerPlant object and one of your own custom objects.

This philosophy of extensibility is really a reflection of the overriding design of all the CodeWarrior tools. In the future, new tools should bring this same functionality to Windows desktop development and Java. The goal is to make application development as simple and easy as possible by giving you powerful tools you can easily retarget and use on the platform of your choice.

Hands-On Exercises

In this exercise you are going to open an existing Constructor project and make some changes to it. Along the way you'll encounter each of the principal resource editors: text traits, layouts, menus, and bitmaps.

You cannot use the resulting resource file without a real application to go along with it, and without a Mac OS computer on which to run the application. You'll work with resources and build a Mac OS application in Chapter 14. Nevertheless, this exercise gives you a feel for the tool, and for what you can do with it. As Constructor adds new targets, these same skills will prove even more useful.

1. Launch Constructor and open a project. Locate and launch Constructor. If you used the standard installation, you'll find Constructor listed among the applications and tools in the Metrowerks CodeWarrior

folder in the Start menu's Programs folder. After you launch it, use the Open command and navigate to the Start Code folder for this exercise (inside the Exer11 folder) and open the SlideShow.rsr file. When you do, the project window appears and should look like Figure 11.24.

FIGURE 11.24

The project window for this exercise shows you existing resources.

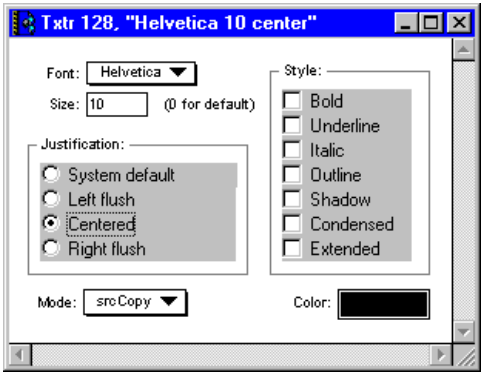
2. Create and edit a text traits resource. The text traits resource is perhaps the simplest kind of resource, so it's a good place to begin. In the project window, select the gray header bar for the text traits resource group, and issue a New Resource (Ctrl+K) command. Then open the group to see the new untitled resource that appears in the list. You can edit the name to be Helvetica 10 Centered if you wish—a clue to what you'll be doing in the rest of this step.

To open up the text traits editor window, double-click the new resource. Use the editor to set the features of this text traits resource to match those shown in Figure 11.25. Use Helvetica, set it to 10-point size, and use centered justification. You can also change the color to blue if you'd like.

If you don't have Helvetica on your system, just pick a different font. Keep in mind, however, that font selection is an important issue in cross-platform development. Remember that this particular tool—Constructor for PowerPlant—targets Mac OS. So you should use fonts

FIGURE 11.25

A text traits resource using the Helvetica font



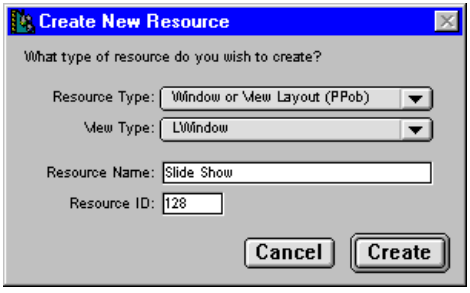
that will be available on the Mac OS machine running the final product. Sticking with basic and standard Mac OS fonts such as Helvetica, Courier, and Palatino is a good practice to follow.

When you're all done, just close the text editor window.

3. Create a layout (PPob) resource. Back in the Constructor project window, select the Windows and Views resource type, and create a new resource. This time a dialog box appears asking you to specify the type of layout resource you want. Set it up to match Figure 11.26. Choose the Window or View Layout resource type, and set the view type to LWindow. Give the resource a name (the solution code calls it Slide Show). The default value for the ID number is fine.

FIGURE 11.26

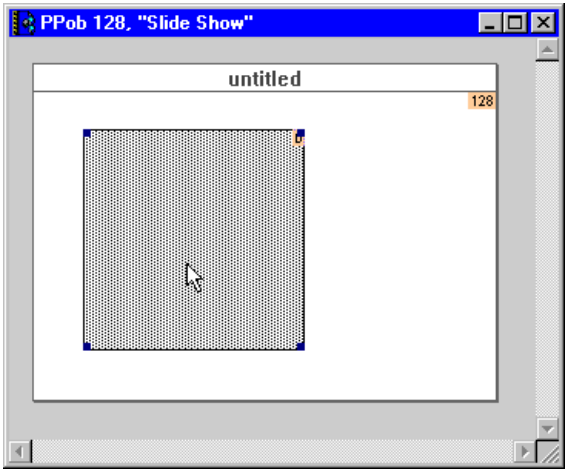
Set up a layout view to match these values.



When you click the Create button, the resource appears in the list in the project window. Use the disclosure triangle to open up the Windows and Views section of the project window, and double-click the new resource to open it so you can begin to add items to the layout.

4. Add an item to the layout. In this step you add an item to the layout. If the Display Classes window isn't open, begin by opening that window. You'll find the command in the Windows menu. Once the window is open, locate the entry for the LPicture class (this is an object that displays a picture), drag it onto the layout, and drop it there. Figure 11.27 shows the result of the work in this step. Use it as a rough guide for placing the item, although precision is not critical here.

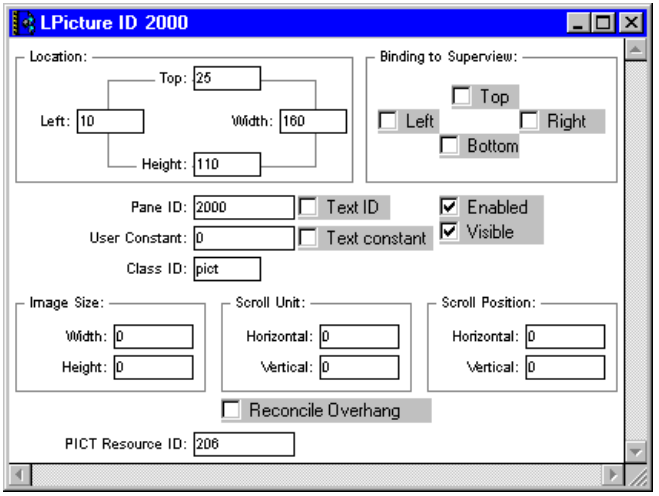
FIGURE 11.27
A layout containing a simple picture



Right now there's nothing in this item, so its contents are rendered as gray. You'll change that in the next step.

5. Modify item properties. In this step you modify the properties of a picture object. Double-click the LPicture object in the layout to open the property inspector window. Set the values to match Figure 11.28. Set the top to 25, the left to 10, the width to 160, the height to 110, the pane

FIGURE 11.28
Set properties for a picture object to modify its location or appearance.



ID to 2000, and the PICT resource ID to 206. This last value is the most important for this step, because it determines what picture is shown inside the object. After you set the value, you should see a picture appear in the layout window.

Feel free at this point to do a little experimenting. For example, the project has pictures with resource values from 206 through 210. You can change the PICT resource ID to display a different picture.

When you are through exploring, close the property inspector for the picture, and close the layout window. The point is to get you accustomed to working with Constructor, not design an entire application. The solution code layout contains more items from a complete application design.

6. Work with menus. In this step you create a new menu in its entirety. To begin, return to the Constructor project window, open the Menu Bar section to see the single menu bar resource, and then double-click that resource to open the menu bar editor. The menu bar already has an "apple" menu, the File menu, and the Edit menu.

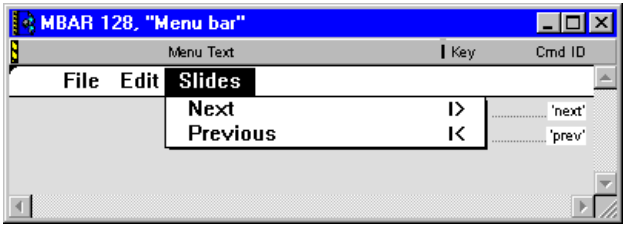
First, create a new menu with the New Menu option (Ctrl+M) in the Edit menu. When it appears, edit the title of the menu to be Slides. Then create two menu items for this menu using the New Menu Item command (Ctrl+K) in the Edit menu. Name one item Next, and the other item Previous. Assign command keys to each item, using the > and < symbols for Next and Previous respectively.

Finally, enter command IDs for each new menu item. You can use a numerical value of 1000 or greater (PowerPlant reserves values less than 1000). Many programmers prefer to use a text-based ID, because it's easier to associate that command with a particular menu item. To enter a text-based ID, activate the command ID number for editing, and then Alt+click the item. The pop-up menu that appears (as shown back in Figure 11.19) lets you switch between numbers and text entry. The solution used the values "next" and "prev" for these two commands. Clearly it's a lot easier to figure out what the "next" command does than trying to recall what command ID "1000" does. It's also less likely that you'll use the same command ID twice if you use text this way.

Figure 11.29 shows you what the menu bar and the new menu will look like after you finish this step.

FIGURE 11.29

The application's menus after creating a new menu with two items



When you're all done, you can close the menu bar editor window. You have just created a complete menu, and completed this exercise.

As always, now is the time to head out on your own, and you have a lot of territory you can explore. For example, you might add a few LCaption objects to the layout, and use the text traits resource you created back in step 2. Add other kinds of objects and explore their property windows. You can work with the bitmap editor and explore the effect of transparency on paste operations. You can work with menus—drag a menu item to a new location, even in a different menu, and see how easy it is to modify the menus. As I mentioned before, the solution code for this exercise has additional items in the layout resource that you can use for further exploration.

When you're all done, you can exit Constructor.

Summary

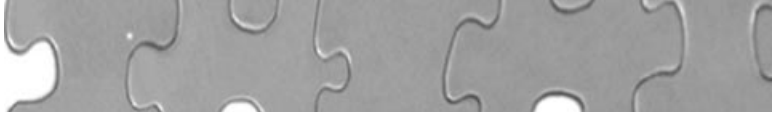
When it comes to Mac OS programming, the PowerPlant/Constructor combination makes a powerful team for rapid application development. This chapter introduced you to Constructor—a GUI-building tool currently targeted at Mac OS. You saw how it uses the same project metaphor to manage resource editing so you can design and create a complete user interface quickly and easily. You also explored each of the principal editors—layouts, menus, bitmaps, and text traits.

Having completed this chapter you have reached another major milestone in your journey. This chapter finishes Part IV on the advanced programming tools that come with CodeWarrior. You have encountered libraries and frameworks written in all the languages and designed for all the targets, and you met the beginnings of a targetable RAD tool in the Windows-hosted version of Constructor for Mac OS—a pretty neat trick when you stop to think about it.

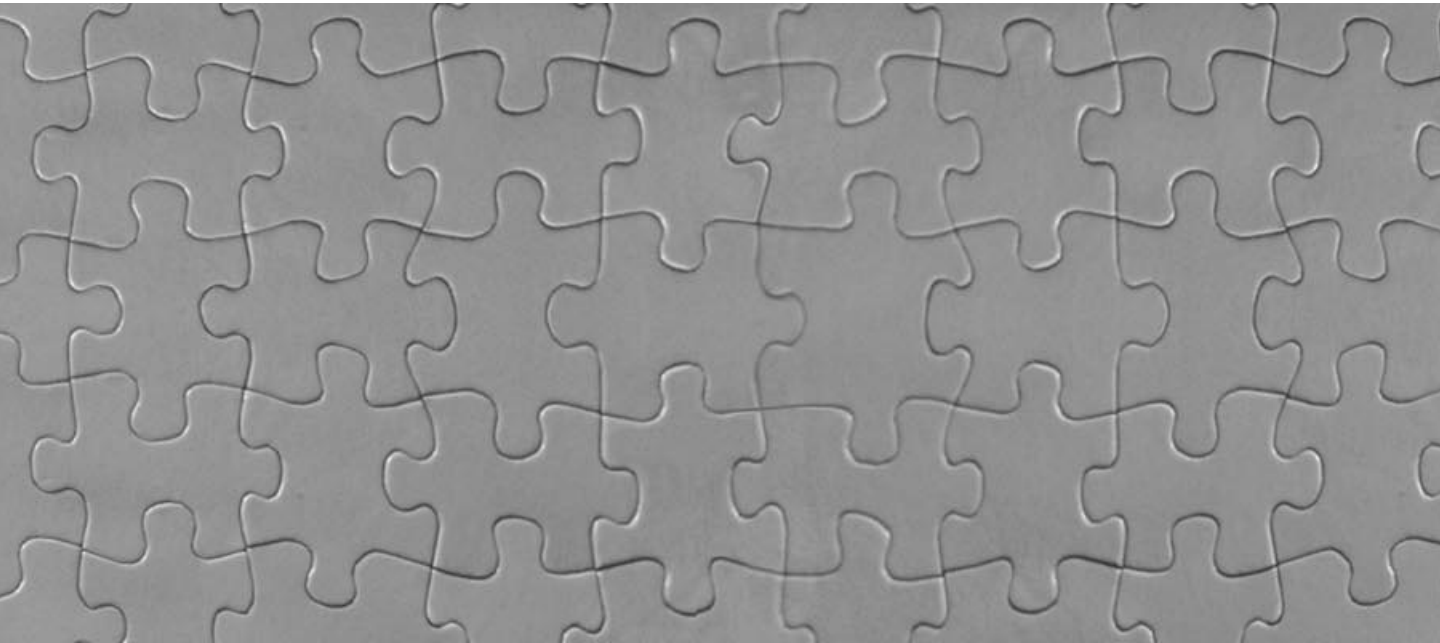
So take a deep breath, sit back, and relax. The adventure is drawing to a close, and the harbor is in sight. One more task remains. To really master the tools you must understand the intricacies of writing code

for a particular target. With a few exceptions (as in this chapter), everything we have talked about until now has been primarily a generic description of how CodeWarrior works for any language and any target. As you begin to actually write code for a particular target, you will encounter new issues. What should the code generation settings be for Windows programming? How do you build a DLL for Windows, or a shared library for Mac OS? How do you test an applet you're building in CodeWarrior?

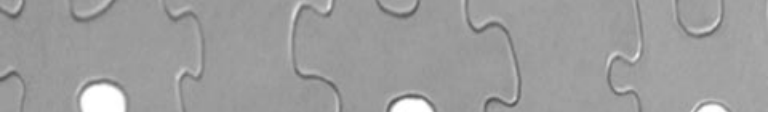
In the remaining chapters we take a detailed look at these and other issues as we discuss how to write code that targets Windows, Java, and Mac OS using CodeWarrior.



PART



V



Hitting the Target

All the ingredients for success are now in your hands. In the previous parts of this book you learned all about CodeWarrior—the great tools and the great source code that comes with them. All that remains is for you to merge those ingredients into a cohesive whole. That's what Part V is all about—using the resources that CodeWarrior provides to write real code for real targets. This is where it all comes together.

Chapters 12 through 14 are each devoted to a particular target—Windows, Java, and Mac OS respectively. In each chapter you'll build a complete project from scratch, and explore some of the target's unique features. In Chapter 15 we'll review where we've been and take a very brief peek at where CodeWarrior is going tomorrow.

After completing Part V, you'll have everything you need to become a true code warrior—the arrow of time and effort you shot when you opened this book will have reached its target.

CHAPTER 12

TWELVE

Programming for Windows

- Setting Up for x86 Development
- Creating a DLL
- Creating an Application

This chapter introduces you to programming for the Windows target running on the x86 family of chips. We'll discuss the unique settings available to you as you program for Windows, and then put everything you have learned to work as you create a dynamically linked library first, and then create an application that uses the library.

Except for the discussion of settings available to you for x86 development, this whole chapter is a hands-on exercise in how to use CodeWarrior for Windows development. This chapter is not going to teach you how to program for the Windows API—that's a subject far beyond the scope of this book.

TIP

CodeWarrior Gold comes with a variety of learning tools you can use to teach yourself Windows programming, including extensive example code and several books in electronic form for both beginning and experienced programmers, in both C and Pascal. It also includes *From Mac to Windows*, by Steve Chernicoff, for Mac OS programmers using CodeWarrior to program for Windows. This book helps you capitalize on your Mac OS experience by highlighting the differences and similarities between these two operating systems.

Setting Up for x86 Development

As you know, CodeWarrior is a multitarget environment. When you create a new project, the first thing you have to do is pick a target. We discussed how to do that back in the "Picking a Target" section of Chapter 3. You go to the project settings, click the Target tab, and select the Win32 x86 Linker as your target.

When you do, the panels available to you for project settings change to reflect your new target. The panels specific to x86 development are:

- x86 Project
- x86 CodeGen
- IR Optimizer
- x86 Linker
- WinRC Compiler

Let's look at each very briefly, so you'll understand the options available to you. Remember that when you use stationery as the source for a new project, all of these settings will have appropriate values. If you find that the stationery that comes with CodeWarrior does not suit your particular needs, you can create your own, as we discussed back in Chapter 3.

x86 Project Settings

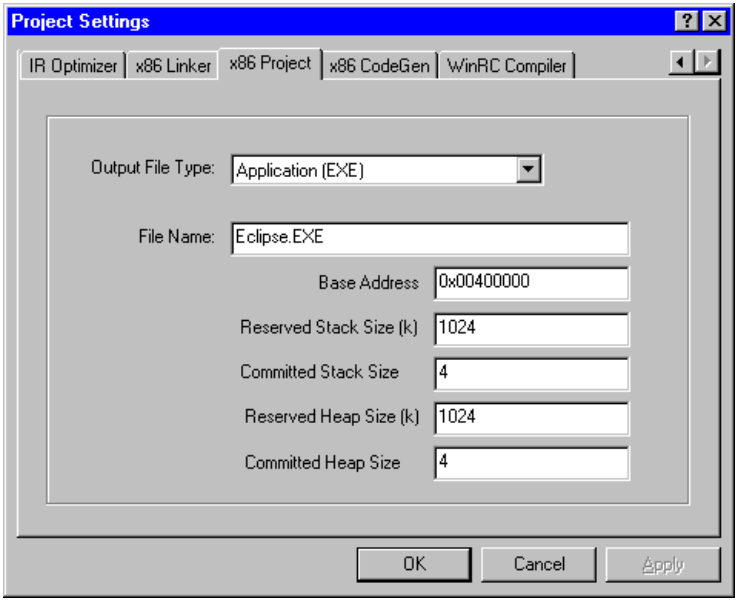
Figure 12.1 illustrates this panel and the options available to you in it. The most significant options are:

- **Output File Type**—choose from application, DLL, or static library
- **File Name**—specify the name of the final program

Even if you use stationery, you'll need to come here to set the name for your final product. Stationery usually sets the Output File Type appropriately. After picking the type of project and setting the name, you're just about done. If the Output File Type is static library, those are the only options.

If you are building an application or DLL, however, you can modify five more settings. Default values usually serve just fine. These additional options are as follows.

FIGURE 12.1
Use the x86 Project settings to determine the type of project you are working on.



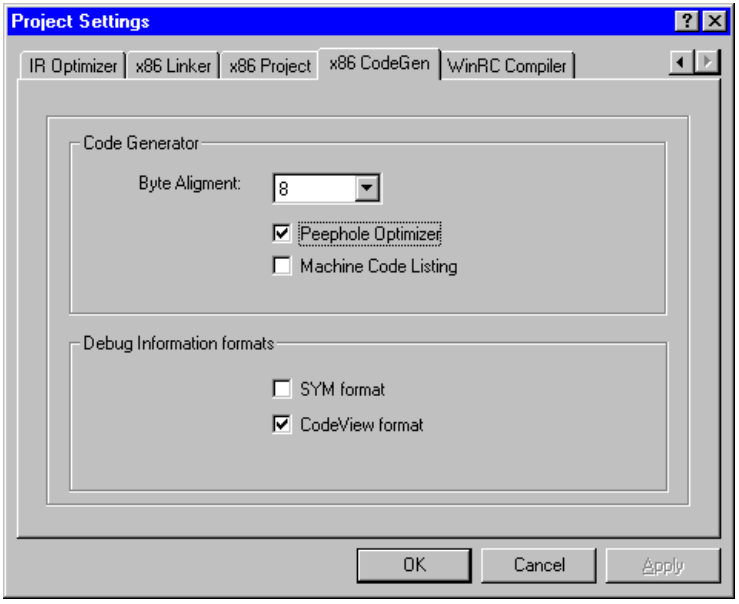
- **Base Address**—the preferred address for loading the application or DLL. When you choose the Output File Type, this value is set to an appropriate default value, 0x00400000 for an application running on Windows 95, 0x10000000 for a DLL.
- **Reserved Stack Size**—the amount of the virtual address space reserved for the stack. The default is 1024K.
- **Committed Stack Size**—the amount of memory initially allocated to the stack on loading. The amount of memory devoted to the stack will increase at runtime as necessary, up to the reserved stack size. The default is 4K.
- **Reserved Heap Size**—the amount of the virtual address space reserved for the dynamic memory heap. The default is 1024K.
- **Committed Heap Size**—the amount of memory initially allocated to the heap on loading, but which increases dynamically at runtime as necessary. The default is 4K.

x86 CodeGen Settings

This panel controls how the x86 back-end compiler—the code generator—works. Figure 12.2 shows you the options in this panel. Once again, stationery usually takes care of all these, but you may wish to modify some or all of them as you work. They are:

- **Byte Alignment**—specifies how structure and class members are aligned in memory. If you choose any value other than one, and the natural byte size of a member does not match the byte alignment you have specified, additional "pad-bytes" are added so that the next member aligns on the specified byte. This can make code work faster because most chips are optimized to retrieve data on even byte boundaries. Padding also means that members will probably take up more room in memory. The default value is to align on 8-byte boundaries.

FIGURE 12.2
Use the x86 CodeGen settings to control the back-end compiler for x86 chips.



- **Peephole Optimizer**—when on, the compiler looks for specific sequences of instructions and replaces them with more efficient instructions. This optimization does not affect debugging and is usually on by default.
- **Machine Code Listing**—when on, the compiler creates a file for each source file in the project that contains the object code generated for that source file. The new file is named the same as the source file, with the .asm extension. This is similar (but not identical) to the information you can see when you disassemble a file.
- **SYM Format**—when on, the compiler generates debugger symbolics in SYM format.
- **CodeView Format**—when on, the compiler generates debugger symbolics in CodeView format.

IR Optimizer Settings

For the x86 target, CodeWarrior can optimize the Intermediate Representation (IR), before the back-end compiler ever sees the code. Figure 12.3 shows you the options that you have for optimizing your code.

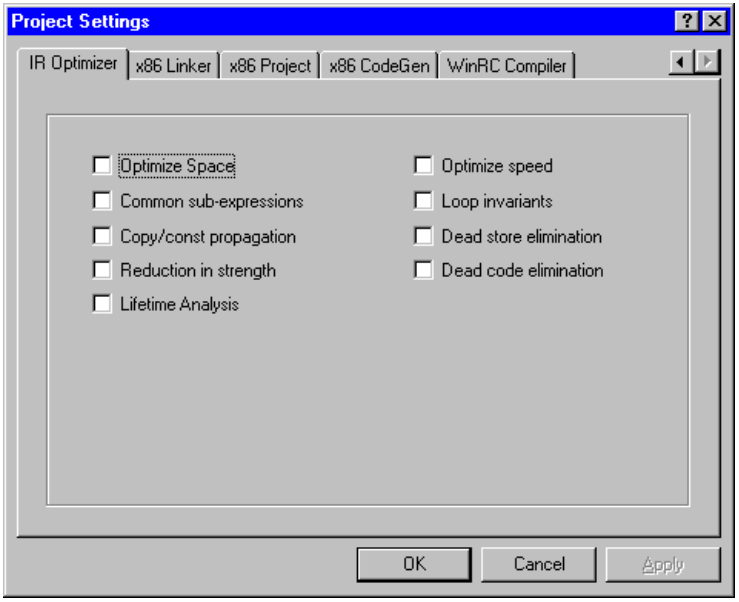
WARNING Remember that if you apply optimizations you may not be able to debug your code because you are likely to lose the one-to-one correspondence between source code and object code that the debugger requires.

Of general importance, you have settings to:

- **Optimize Space**—perform optimizations to reduce the size of the object code.
- **Optimize Speed**—perform optimizations to increase the speed of execution of the object code.

If both settings are on, the compiler performs optimizations that have no significant negative impact on either size or speed. If only one

FIGURE 12.3
Use the IR Optimizer settings to streamline your object code.



of these settings is on, the compiler will optimize for that factor—either space or speed—without regard to the effect on the other factor.

These two settings affect not only the IR, but also tell the back-end compiler to perform certain optimizations as well. For example, if you are optimizing for speed only, the back-end compiler aligns functions on a 16-byte boundary (at some cost in size). If you are optimizing for size and the compiler detects an identical sequence of instructions, it deletes the sequence and inserts a jump to the original sequence (at some cost in speed).

With respect to the IR, these two options (size and speed) do not choose particular optimization strategies. They simply control how the optimizing compiler applies the other specific optimizations you choose in this panel. If both of these options are off, no optimization takes place. If one or the other or both are on, then the compiler uses whatever strategies you choose in the rest of the panel. Table 12.1 lists each additional option and its usual effect on the final object code.

TABLE 12.1 The effect of IR optimizations on code size and speed

Type	Effect on Size	Effect on Speed
Common SubExpressions	smaller	faster
Copy/Const Propagation	smaller	faster
Reduction in Strength	slightly larger	faster
Lifetime Analysis	smaller	faster
Loop Invariants	slightly larger	faster
Dead Store Elimination	smaller	faster
Dead Code Elimination	smaller	none

x86 Linker Settings

The linker puts all the object code together in a single package. Your code starts executing somewhere, and that is the entry point you specify in this panel. Figure 12.4 shows all the options, arranged in three groups: Entry Point, Subsystem ID, and Output settings.

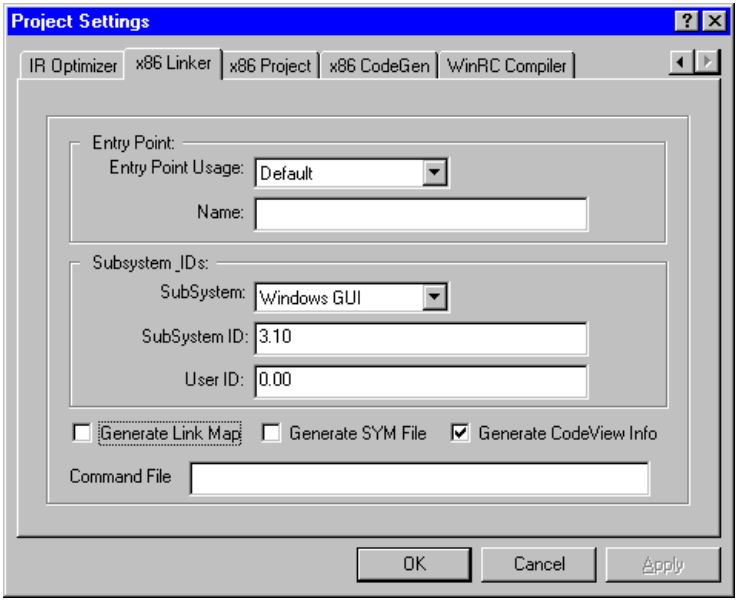
The options for Entry Point Usage (a pop-up menu) are:

- **Default**—sets entry point correctly for console applications, Windows applications, and DLLs.
- **User Specified**—uses the name in the Name field of this panel.
- **None**—for DLLs with no initialization routine.

The options for SubSystem (another pop-up menu) are:

- **Windows GUI**—for a Windows application, that is, one with a **WinMain()** entry-point routine.
- **Windows CUI**—for a console application running under Windows, that is, one with a **main()** entry-point routine.
- **Native**—for drivers and system executables.

FIGURE 12.4
Use the x86 Linker settings to control how your code is put together.



- **Unknown**—tells the linker to determine for itself, based on the type of **main()** function.

After you have specified the subsystem, you set the SubSystem ID. The value 3.10 is appropriate for all Windows programming. The User ID is a number you can provide if you wish. The linker puts this number (essentially a version number) in the header of your project's output file.

In almost all cases, project stationery takes care of all these settings appropriately. The only values in this panel you are likely to change on your own are the Output settings. They are:

- **Generate Link Map**—tells the linker to generate a link map.
- **Generate SYM File**—tells the linker to create a SYM file for debugging.
- **Generate CodeView Info**—tells the linker to put CodeView symbolics in the final output file. This is the preferred setting when you want to debug x86 code.

- **Command File**—the name of a file that contains additional commands to control the operation of the linker—for example, a list of symbols in a DLL you want the linker to export. This field is usually empty.

WinRC Compiler Settings

This panel controls the resource compiler and is very simple. It has a single entry, where you provide a file name. This optional file is analogous to the project prefix specified in the C++ Compiler panel. It would typically declare the values of any constants or terms used in the resource file, and might also include preprocessor directives. You can even use precompiled resource headers such as the Win32ResHeaders++ file that comes with CodeWarrior, The Metrowerks resource compiler is compatible with any standard resource file for Windows programming.

You can also precompile your resources into a single file (with the extension .res) and add that file to your project. The WinRes Import plug-in tool reads in the data, and the linker combines it into your final output.

Other x86 Features

Before we get started on writing actual code, let's look at a few other features of programming for the Win32 target in CodeWarrior.

The Metrowerks compiler and linker support the standard COFF object format and calling conventions currently in use in Windows programming (with one exception discussed in a moment). As a result, code generated by CodeWarrior is compatible with other code and tools in the industry. For example, the Metrowerks linker can link in object code created by Visual C++. You can debug a CodeWarrior-created executable with third-party debuggers that support CodeView

symbolics. You can mix and match DLLs created by Visual C++ and CodeWarrior. The CodeWarrior Pascal language supports Borland's Turbo Pascal language extensions.

The areas of compatibility with industry standards are widespread, but not universal. CodeWarrior is incompatible with other tools (and they with each other) in one area, and that is in C++ classes that use multiple inheritance. If your code uses multiple inheritance, then code generated by other compilers will not be compatible with CodeWarrior-generated code. No industry-wide standard has yet emerged for this feature of the language, and as a result code generated by different compilers is incompatible.

With respect to the C/C++ language, CodeWarrior also implements a variety of Windows-oriented **#pragma** statements. The Metrowerks standard C library has some special Windows-related functionality, mostly to support the Microsoft Foundation Classes. Once again, these features are aimed at supporting industry standards and enhancing the ease with which you can move code from one environment to another.

The *Targeting Win32* manual that comes as part of the CodeWarrior documentation has all the details on these and other aspects of programming for Windows using CodeWarrior. You'll also find details of the x86 inline assembler built into the C/C++ compiler for x86 development. You should also examine the *C, C++, and Assembly Language Reference* that comes as part of the documentation because it describes x86-specific features of the C/C++ language front-end compiler.

Creating a DLL

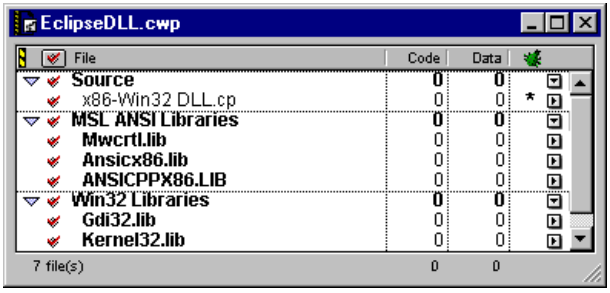
In this exercise you'll create a complete DLL from scratch, ready to use and debug. In the other two exercises in this chapter, you'll build an application that uses the DLL, and then debug both the application and the DLL.

1. Create a new project. If CodeWarrior is not already running, launch CodeWarrior, and then choose the New Project item (Shift+Ctrl+N) from the File menu to create a new project. When the New Project dialog box appears, do two things:

1. Turn *off* the Create Folder check box
2. Select the C++ DLL stationery

When you click the OK button, the standard dialog box appears so you can save the new project. Name the project EclipseDLL.cwp, and save it inside the Start Code folder for this chapter, inside the Exer12 folder. When the new project window appears, as shown in Figure 12.5, notice that it includes a placeholder source file named x86_Win32_DLL.c. You'll be replacing this file with new source code in a subsequent step.

FIGURE 12.5
The new DLL project ready to go



2. Modify and examine project settings. In this step you do a little setup work for the project. Although you used stationery to begin, stationery can't do everything. You'll turn on debugging and examine a few other features already set by stationery.

Go to project settings and examine the x86 Project settings panel (see Figure 12.1 earlier in this chapter). Notice that the output file type is already a dynamic link library. If necessary, change the name of the output file to Eclipse.dll.

Next, go to the x86 CodeGen settings panel (see Figure 12.2). If CodeView format is not activated, turn it on. In a subsequent exercise in this chapter you'll be debugging this library.

Finally, go to the x86 Linker settings panel (see Figure 12.4). Change the Entry Point Usage to None because this DLL does not require initialization and does not have a **main()** function. The default settings for the Subsystem are fine. Because you'll be debugging later on, make sure that the Generate CodeView Info option is on.

TIP

To turn on CodeView symbolics generation, you can also choose the Enable Debugging item in the Project menu.

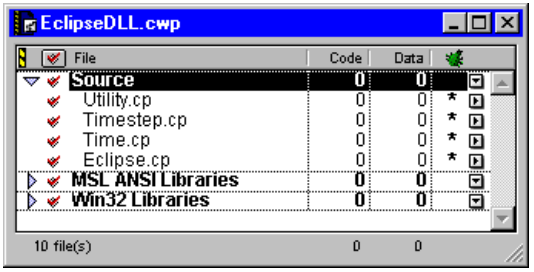
When you are all through examining these three settings panels, click the OK button to accept any changes and modify the project settings.

3. Add source files to the project. In this step you add the necessary source files to create the DLL. To begin, select the Source group name so new files will be added into the Source group. Then choose the Add Files item in the Project menu, and in the resulting dialog box navigate to the Exer12\Start Code\DLL Source directory. From that directory, add the following four source files to the project:

- Eclipse.cp
- LumTime.cp
- Timestep.cp
- Utility.cp

After clicking the Add button to add the files, return to the project window. Select the x86_Win32_DLL.c placeholder file, and remove it from the project. Use the Remove Files option in the Project menu. When you're all through, the project window should look like Figure 12.6.

FIGURE 12.6
The DLL project with all the correct files in place



NOTE

Although the source code in these files is straight C code, the file name extension of .cp invokes the C++ compiler. The application you build in the next section will use the MFC and C++, and the resulting name mangling of function calls to the library requires that the library function calls match. If they do not, you'll get link errors when you try to build the application.

4. Export functions from the DLL. For other code (such as the application you create in the next exercise) to use routines in the DLL, you must export functions.

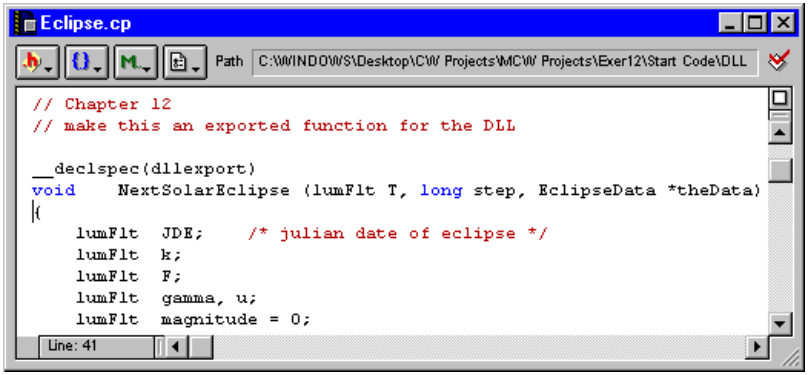
In the project window, double-click the Eclipse.cp file to open it in the editor. Use the function pop-up menu to locate the **NextSolarEclipse()** function. You'll see a comment there that indicates you're in the right spot. This is a function you'll be calling from outside the library so it needs to be exported. To export this function name, precede the function with the declaration modifier:

```
__declspec(dllexport)
```

Figure 12.7 shows you the actual source code after you add the modifier in the right place.

After you have made this modification to the code, do the exact same thing for the **NextLunarEclipse()** function in the same source file. Locate the function and add the declaration modifier to export the function name.

FIGURE 12.7
Export the **NextSolarEclipse()** function so applications can use it.



You have now exported two functions. You can save your work and close the window. You're all done.

The source code for LumTime.cp and Timestep.cp already exports a few functions as well. If you wish, you can open these files and examine the functions to see which are exported.

TIP

The *Targeting Win32* manual in the CodeWarrior documentation covers the details of declaration modifiers and other ways to export functions and variables in a DLL.

5. Build the DLL. To actually create the DLL, choose Make from the project menu. CodeWarrior compiles the code and links the final output. In this case it creates two files, the actual Eclipse.dll library, and a file named Eclipse.dll.lib. This second file is what you add to an application project when you want it to use the functions inside the DLL. You can go to the Windows desktop and examine the Start Code folder to see both files if you wish.

Congratulations! You've built a DLL with CodeWarrior. Now let's put it to use.

NOTE

If you followed the steps carefully, you should have no problems. However, if you encounter errors in the build process, go back through the steps in this exercise to see where you might have strayed from the instructions. If all else fails, you can go to the Solution Code folder for this exercise and find the necessary files there, compare them to your work, and figure out what went wrong.

Creating an Application

In this part of the exercise you are going to use the MFC to create a simple application that uses the DLL you just created to calculate and display the date for either a solar or lunar eclipse. The goal here is not to create a pretty interface but to show you how to use CodeWarrior to build both a DLL and an application. You're already half-way toward the goal—you've got the DLL. Now let's put it together with an application project.

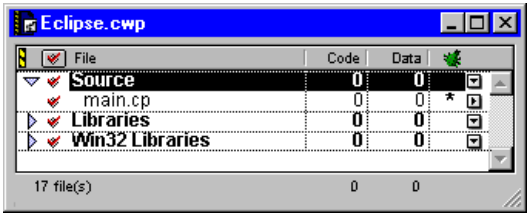
1. Launch CodeWarrior and create a new project. Launch CodeWarrior, and then choose the New Project item (Shift+Ctrl+N) from the File menu to create a new project. When the New Project dialog box appears, do two things:

1. Turn *off* the Create Folder check box
2. Select the MFC Application stationery

When you click the OK button, the standard dialog box appears to save the new project. Name the project Eclipse.cwp, and save it inside the Start Code folder for this chapter, inside the Exer12 folder. When the new project window appears, as shown in Figure 12.8, notice that it includes a placeholder source file named main.cp. You'll be replacing this file with new source code in a subsequent step.

2. Modify and examine project settings. This is just like step 2 for the DLL. You'll make sure debugging is turned on and examine a few other features already set by stationery. You'll examine the x86 project, x86 CodeGen, and x86 Linker settings.

FIGURE 12.8
The new application project is ready to go.



Go to project settings and examine the x86 Project panel (see Figure 12.1 earlier in this chapter). Notice that the output file type is already an application, as it should be. If necessary, rename the output file to Eclipse.exe.

Next, go to the x86 CodeGen panel (see Figure 12.2). If CodeView format is not activated, turn it on. In a subsequent exercise in this chapter you'll be debugging this library.

Finally, go to the x86 Linker panel (see Figure 12.4). In this case the default settings (notice the entry point and the Subsystem) are fine. Because you'll be debugging, make sure that the Generate CodeView Info option is on.

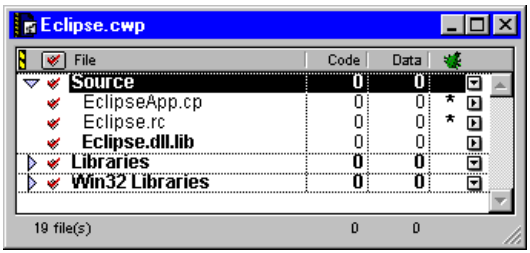
When you are all through examining these three panels, click the OK button to accept your changes and modify the project settings.

3. Add the necessary source files. In this step you add the necessary source files to create the application. To begin, select the Source group name so new files will be added into the Source group. Then choose the Add Files item in the Project menu, and in the resulting dialog box navigate to the Exer12\Start Code directory. From that directory, add the following files to the project:

- **EclipseApp.cp**—the source code for the application.
- **Eclipse.rc**—the resources for the application.
- **Eclipse.dll.lib**—the exported functions to link with the application.

After clicking the Add button to add the files, return to the project window. Select the main.cp placeholder file, and remove it from the project. Use the Remove Files option in the Project menu. When you're all through, the project window should look like Figure 12.9.

FIGURE 12.9
The application project with the correct files in place



4. Examine the code that calls DLL functions. The code for the complete application has been provide for you. Open the Eclipse- App.cp file (and the Eclipse.rc file if you wish) to explore. The application has a Calculate menu with four items to calculate the next or previous solar or lunar eclipse from a starting date of January 1, 1994.

In particular, look for the **EclipseWindow::CalcSolar()** function. You'll see a series of function calls to routines in the DLL. They are:

- **TFromUniversal()**
- **NextSolarEclipse()**
- **JulianToUniversal()**

You'll see similar calls in the **EclipseWindow::CalcLunar()** function as well. By separating the calculation engine from the interface code, you can reuse the engine in other applications, and even for other platforms. The source code you compiled in the DLL could become a shared library on Mac OS with no change whatsoever.

5. Build and run the application. Choose the Run item from the Project menu, and CodeWarrior compiles, links, and launches the applica-

tion. It takes a few moments as the project manager locates and loads all the libraries, but when the process is complete Eclipse.exe appears on your screen.

Choose an item from the Calculate menu and watch what happens—data about the next or previous eclipse appears. Excellent! You have successfully created an MFC-based application that uses a DLL. Of course, in your own work you'll need to debug both of these items. And that's what you'll explore in this chapter's final exercise. You can quit CodeWarrior if you wish—you don't need to have it running while debugging.

Debugging x86 Code

In this final exercise you learn how to debug a DLL. Everything you learned in Chapter 8 on the debugger applies. No unique or special considerations are involved when working with x86 code. Nevertheless, it is helpful to actually practice how to debug not only standard application code, but a DLL as well.

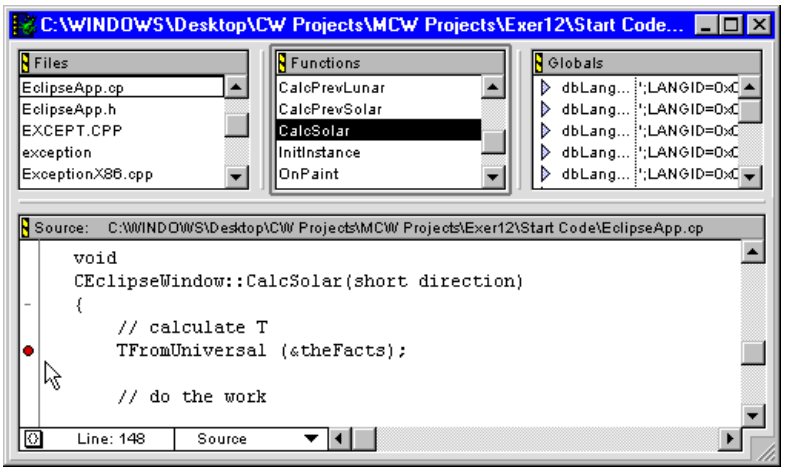
1. Launch the debugger. Launch the CodeWarrior debugger, and when the dialog box appears asking you to open a file, navigate to the Exer12\Start Code folder and open Eclipse.exe. In response, the debugger opens the application, reads the CodeView symbolics information inside, and launches the application preparatory to running it under debugger control.

After a moment you'll get a second dialog box asking you to locate a file named APPMODUL.CPP. The debugger wants to display the source code for this file, which is where an MFC application begins execution. This is an MFC source file located in the CodeWarrior\Win32-x86 Support\(MFC 4.2)\MFC 4.2 Source directory on a full CodeWarrior installation. It may not be available with a CodeWarrior Lite installation. If you do not have this file, you can simply cancel the dialog box asking you to locate APPMODUL.CPP. You'll see assembly

language code in the debugger's Program window instead of source code, but that's not going to be a problem for you.

2. Set a breakpoint in EclipseApp.cp. So you can study what's going on in both the application and the DLL, you'll set a breakpoint to stop at a good spot. Switch to the debugger's browser window, select the EclipseApp.cp file in the File pane, and then the **CalcSolar()** function in the Function pane. The source code appears in the Source pane, and you can set a breakpoint at the call to the function **TFromUniversal()**. Figure 12.10 shows the Browser window after you have set the breakpoint.

FIGURE 12.10
Set a breakpoint at the call to **TFromUniversal()** in CalcSolar().



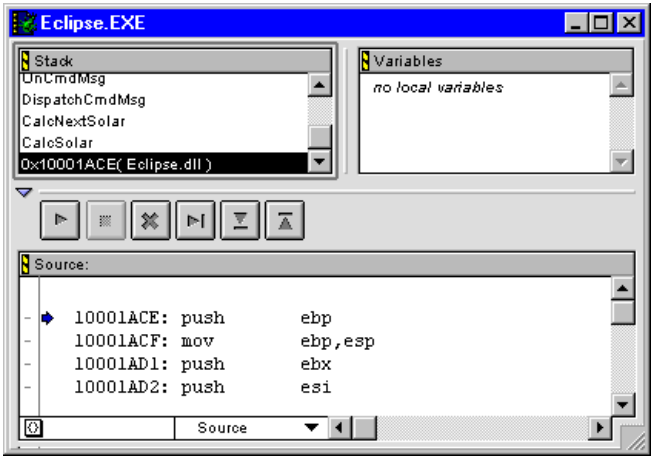
3. Run the application. Issue a Run command from the debugger to resume execution. The Eclipse.exe program appears. When it does, choose the Next Solar Eclipse item from the Calculate menu, so you'll stop at the breakpoint you set in the previous step.

4. Trace into the TFromUniversal() routine. Once you have stopped at the breakpoint, issue a Step Into (trace) command to step into this

function. The program window's Source pane shows you a bit of assembly code—this is a jump table into the DLL functions.

Issue a second Step Into command to jump to the actual DLL function. Figure 12.11 shows the result—more assembly code! You'll notice in the Stack pane that you are in fact inside the DLL, but you can't see the source code.

FIGURE 12.11
The assembly language code for **TFromUniversal()** in Eclipse.dll.



The reason you can't see the source code is that the debugger has not opened up the symbolics information for the DLL. If you do that, you'll be able to see the source code for the DLL.

Kill the running application to ensure that the debugger can start fresh. The Program window for Eclipse.exe disappears, but the Source Browser window remains.

5. Open symbolics for the DLL. Use the debugger's Open command in the File menu, and in the resulting dialog box navigate to the Start Code folder for this exercise. Locate and open the Eclipse.dll file—the symbolics information is inside the DLL itself.

Now the debugger has access to the information it needs to display source code for you.

6. Run the application again. After opening the DLL symbolics, a browser window appears for the DLL (in addition to the browser window that exists for the application). If you want, you can set a breakpoint directly in the DLL code by using the DLL's browser window. There's no need to do that here, but feel free to do so if you wish.

Switch to the application's browser and issue a Run command. The application launches and the Program window for the application appears, with the application stopped at the beginning. Issue a second Run command to resume execution.

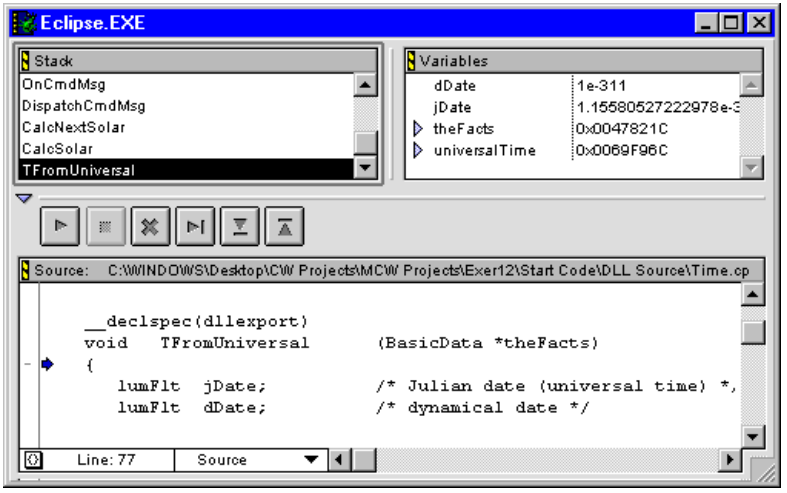
Choose the Next Solar Eclipse item from the Calculate menu, so you'll stop at the breakpoint you set previously.

7. Step into DLL code. Now that you have stopped at the call to **TFromUniversal()** one more time, issue a Step Into command. You'll see the assembly language for the DLL's jump table.

Issue a second Step Into command, and this time the debugger displays the actual DLL source, as shown in Figure 12.12. All the local variables are there, and the full range of debugger features is available to you.

That's it for this exercise, and this chapter. Feel free to explore on your own, or to use this as a starting point for a little more comprehensive Windows programming. For example, you may want to add a dialog box or an edit field in the window to allow the user to specify a starting date for the search. Whatever direction you choose, CodeWarrior provides the support you need to get to your goal.

FIGURE 12.12
The source code from the DLL is available after you open the DLL symbolics.



Summary

This chapter introduced you the intricacies of x86 programming with CodeWarrior. As you can see, the process is pretty straightforward. Writing computer code requires a lot of care to ensure that all the various components are consistent with each other—but that's nothing new. Using CodeWarrior itself is pretty simple.

At this point, no bounds are limiting you. You have all the knowledge you need to use CodeWarrior for Windows programming. However, we still have a bit further to travel in this book. You might want to go somewhere other than Windows programming. In the remaining chapters we'll look at Java and Mac OS as targets for CodeWarrior.

CHAPTER 13

THIRTEEN

Programming for Java

- Setting Up for Java Development
- Creating a Java Applet
- Creating a Java Application

This chapter introduces you to programming for the Java Virtual Machine (JVM) target. I'll discuss the unique settings available to you as you program for the JVM. Then you'll put everything you have learned to work as you create an applet, and then an application. Both of these simple projects will use the same code, but they run under different circumstances. An applet appears within an HTML browser. An application runs within (or under control of) a Java runtime environment.

Except for the discussion of settings available to you for Java development, this chapter is a hands-on exercise on how to use CodeWarrior for Java development.

The Thought behind the Java Virtual Machine

We discussed the JVM briefly back in Chapter 1 in a sidebar titled "What Is a Virtual Machine?" Fundamentally, it is a processor instruction set without the microprocessor. Java source code is compiled into Java bytecode—the JVM instruction set, which is similar to traditional object code created by a C or Pascal compiler. Unlike object code, however, Java bytecode cannot run directly on any real machine. It must be translated (interpreted or recompiled) into instructions the target processor understands.

This translation occurs inside some piece of software that understands Java bytecode on the one hand, and the native instruction set of the target machine on the other. This software can take the form of a Java runtime interpreter, or a just-in-time (JIT) compiler. In either case the bytecode is read, translated, and the resulting native instructions run on the target machine.

continued ➠

The Thought behind the Java Virtual Machine

Here you begin to see a parallel between the Java architecture and CodeWarrior. Remember the elegant concept behind CodeWarrior—languages are compiled into the Intermediate Representation, and then code generators create target-specific instructions. This is the fundamental design principle that gives CodeWarrior its power and flexibility.

Now consider how Java works, and you'll see a close parallel. You write source code in a language (in this case Java) and a language compiler turns that code into Java bytecode. The bytecode is an intermediate representation. Like the CodeWarrior IR, it cannot run directly on any machine because it must go through a further translation process. For CodeWarrior, that's a back-end compiler. For Java bytecode, it's a Java interpreter or JIT compiler.

By making the JVM an abstract machine, the Java designers have used the same design pattern found in CodeWarrior, and gained the same kind of flexibility. The two processes are not identical. For example, the CodeWarrior back-end compiles the IR once, and you can execute the code many times. A JIT compiler translates the bytecode for a Java applet or application every time it runs. A Java interpreter translates each bytecode while it runs, and then forgets it. If code repeats, the bytecode is just translated all over again.

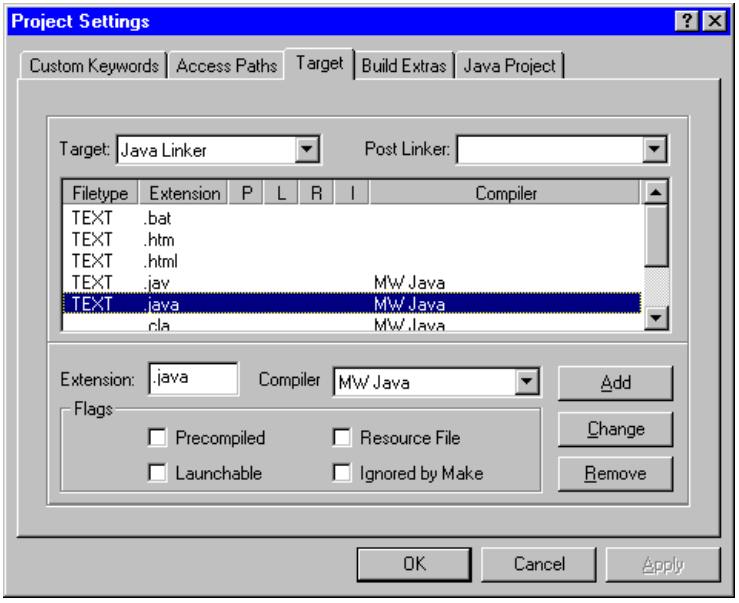
Nevertheless, despite differences in detail, the design pattern is the same: a language front-end, an intermediate stage that is platform-independent, and a back-end that generates code native to the target. Great ideas tend to show up in different surroundings and environments. The concept of the IR is a great idea, and here you can see it in a software development environment and in a virtual microprocessor runtime architecture. Both instances of the idea provide great advantages to you in terms of flexibility and reuse of skills.

Setting Up for Java Development

As with any other target, the first thing you do when creating a new project from scratch (as opposed to creating a project from stationery) is to go to the Target panel and choose your target. We discussed how to do that back in the "Picking a Target" section of Chapter 3. You go to the project settings, click the Target tab, and for Java development you select the Java Linker as your target. Figure 13.1 shows you this settings panel, which includes the file naming extensions available for Java projects.

FIGURE 13.1

Choose Java as a target in the Target settings panel.



As with any other target, once you specify the target, the panels change so that only those that are appropriate for your selected target are available to you. When you choose Java as your target, the C/C++ and Pascal panels disappear, along with a few others. Only one panel is specific to Java development, the Java Project settings.

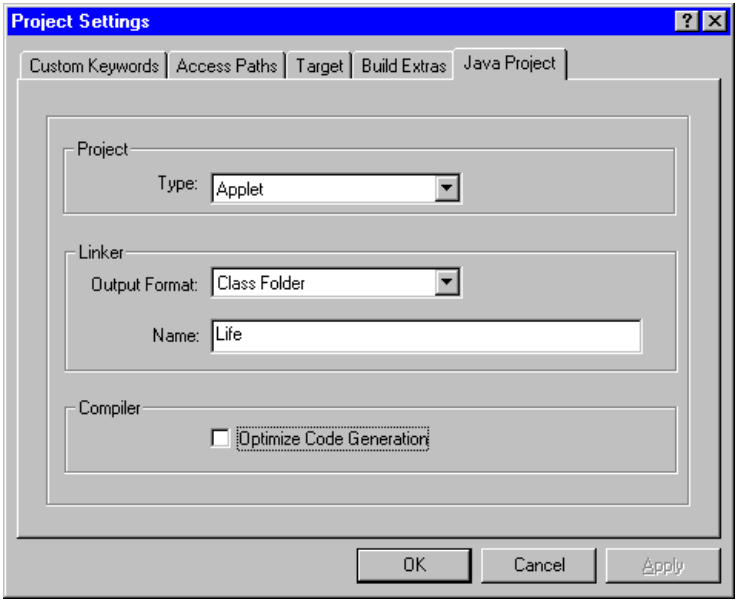
Let's look at this panel briefly, so you'll understand the options available. Remember that when you use stationery as the source for a new project, the settings will have appropriate values. If you find the stationery that comes with CodeWarrior does not suit your needs, you can create your own, as we discussed back in Chapter 3.

Java Project Settings

The Java Project settings panel encompasses all settings for Java: project type, compiler, and linker. Figure 13.2 illustrates the options available in this panel, and you can see that it is divided into three sections. The options are all pretty straightforward. They are:

- **Type**—choose from applet, application, and library
- **Output Format**—choose from zip file or class folder
- **Name**—specify the name of your final zip file or class folder
- **Optimize Code Generation**—apply optimizations to your code

FIGURE 13.2
All Java-related options are controlled from the Java Project settings panel.



Some of these options deserve a little deeper explanation.

When you write Java code, each public class you create must exist in a separate source file. All your source files should end with the extension .java. When you compile your code, each source file becomes a corresponding .class file. The .class file contains the Java bytecode generated by the compiler. Unlike C and Pascal, no object code is stored in the project file. Java object code—the Java bytecode from the Java compiler—is kept in this collection of .class files, one per .java file.

When it is time to actually output the .class files, you tell the linker how you want it to store the files. If you choose the class folder option, CodeWarrior creates a directory in which it places all the .class files. The files in this directory will be organized in subdirectories that match the package structure in your project. If you choose zip file as the output format, CodeWarrior creates a single zip file and places all the .class files inside the zip archive.

NOTE

The .class files in the zip archive are not compressed. This is simply a storage mechanism that allows you to work with all the .class files as a single unit. This can be a lot easier to manage when sending the files over a network, for example.

The Java optimizer optimizes for speed, not size. If you turn on the Java optimizer, the compiler will turn **static**, **final**, and **private** methods into inline functions. This can make your code somewhat larger, particularly if you have many calls to such methods. However, your code will execute more quickly.

As you can see, setting up a Java project is remarkably simple. The design of the Java language and the Java Virtual Machine help make this one of the simplest targets to program for in CodeWarrior.

Creating a Java Applet

In this exercise you create a Java applet that displays the game of Life. Life is a little checkerboard world where each black square is "alive." If a square has too many neighbors, it dies from overcrowding. If it has too few neighbors, it dies from lack of community. If it has the right number of neighbors, it stays alive. If an empty square has just the right number of neighbors, it becomes alive, thus altering the pattern. As squares are born, live, and die, the patterns that form can be very intriguing. Stable patterns can emerge, or an entire population can die out very quickly. Peter Lewis, an independent software developer from Australia and renowned author of some of the best Internet utilities available for Mac OS, wrote the code for this applet and has generously allowed me to use it in this chapter.

Life is an intriguing simulation, but the point here is to build the applet, not get lost in the details of Life. So let's get started.

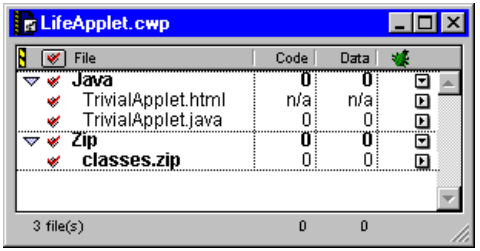
1. Create a new project. If CodeWarrior is not already running, launch CodeWarrior, and then choose the New Project item (Shift+Ctrl+N) from the File menu to create a new project. When the New Project dialog box appears, do two things:

1. Turn *off* the Create Folder check box
2. Select the Java Applet stationery

When you click the OK button, the standard dialog box appears so you can save the new project. Name the project LifeApplet.cwp, and save it inside the Start Code folder for this chapter, which is inside the Exer13 folder. When the new project window appears, as shown in Figure 13.3, notice that it includes two placeholder source files named TrivialApplet.html and TrivialApplet.java. You'll be replacing these files with new source code in a subsequent step.

FIGURE 13.3

This new Java applet project is ready to go.



The presence of the .html file is an indication of how CodeWarrior works with Java applets. In order to run an applet, you must view it inside an HTML browser. The .html file provides the HTML code required for the browser to display the applet. You can open and edit HTML files with the CodeWarrior editor.

2. Modify and examine project settings. In this step you set the name for the final output for the project.

Go to project settings and examine the Java Project panel (shown in Figure 13.2). Notice that the type of project is already set to applet, and the format is a class folder. The default name for the resulting folder is Java Classes.

Change the name of the output folder to Life, and then click the OK button to accept the change and modify the project settings.

3. Add source files to the project. In this step you add the necessary source files to create the applet. You need to add three files, two in the Start Code folder for this project, and one in the Java Support folder inside the CodeWarrior folder.

To begin, go to the project window and select the Java group name (or one of the files in the group) so that new files will be added into this group. Then choose the Add Files item in the Project menu. In the

resulting dialog box, navigate to the Exer13\Start Code directory. From that directory, add the following source files to the project:

- Life.html
- Life.java

After clicking the Add button to add the files, repeat this process to add one more file. When the add file dialog box appears, navigate to the CodeWarrior\Java Support\Java Source directory. From this directory, add the following source file to the project:

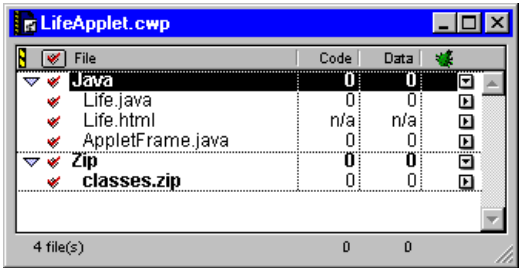
- AppletFrame.java

The AppletFrame.java file is a bit of Java code from Metrowerks that you can use to display an applet in a frame. It instantiates and runs the applet, sets the frame size to fit the applet, and disposes of the applet when the window closes.

After you've added this file, return to the project window. Select the two original placeholder files and remove them from the project. Use the Remove Files option in the Project menu. When you're all through, the project window should look like Figure 13.4.

FIGURE 13.4

The Java applet project with all the correct files in place.

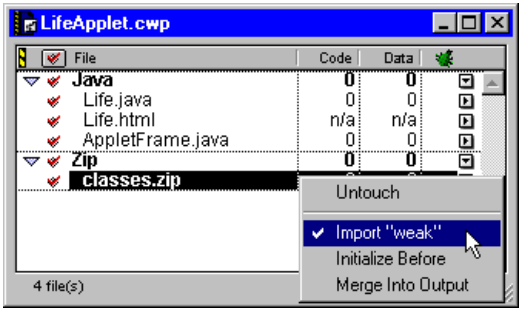


4. Study classes.zip As you learned in Chapter 10, the classes.zip file is the Java library. It contains all the classes in all the packages that are part of the Java language from Sun MicroSystems. You may recall from Chapter 10 that you can work with this file either as a shared library (the equivalent of a DLL) or as a static library.

Click the pop-up menu to the right of the file name, and you'll see that you have certain options with respect to how the linker treats this library, as shown in Figure 13.5. By default, this library has the Import "Weak" option on. This means that the linker does not include the code from classes.zip directly into your project, and this is the typical arrangement when you're programming. However, you must have the file available when the applet runs, just as you would for any DLL.

FIGURE 13.5

You can control how the linker uses classes.zip with the pop-up menu.



When you install CodeWarrior, it places the classes.zip file in the Windows\Java\Classes directory. If you want to add it to your own projects, that's where you should go looking for it.

The other two options are Initialize Before and Merge Into Output.

Initialize Before only has an impact if you are including more than one library, and the loading order is important. The library marked Initialize Before will be loaded and initialized first.

The Merge Into Output option is the opposite of Import "Weak." This option tells the linker to include the actual code in your final build.

WARNING

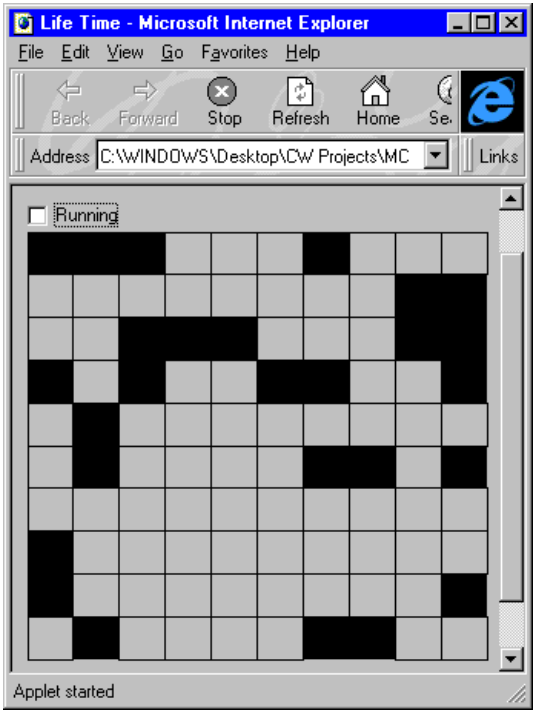
If you are building an applet for distribution on the Internet, you would likely not want to import weak, because you cannot guarantee the presence of classes.zip on any particular machine other than your own.

5. Run the applet. The project is ready to run, so choose Run from the Project menu. This tells CodeWarrior to compile, link, and launch the applet. CodeWarrior tells Internet Explorer 3.0—a Java-savvy browser—to open the first .html file in the CodeWarrior project. In this case, that's Life.html, which is designed to display the applet.

After you issue the Run command, in a few moments Internet Explorer appears, and the applet will be running. Figure 13.6 shows you what it looks like. You can use the Running check box to halt the game, modify the pattern of Life, and start it up again to see what happens. Enjoy the show. Feel free to explore the source code as well.

As you can see, developing and running an applet in CodeWarrior is very simple. Although this exercise has been very direct, don't forget the great tools you have available to you. You can use the editor to write the code, the browser to explore your Java classes, and the debugger to step through your code and watch what's going on. All of these tools work for Java the same as they do for other languages and targets.

FIGURE 13.6
Into every Java applet, a little Life must fall.



Creating a Java Application

Building a Java application is just about as easy as building an applet. In this exercise you'll use the same source files to build a Java application that runs under Jview, Microsoft's Java runtime. You're going to find the steps almost identical to what you do for an applet, with the exception that you set the project up to be an application.

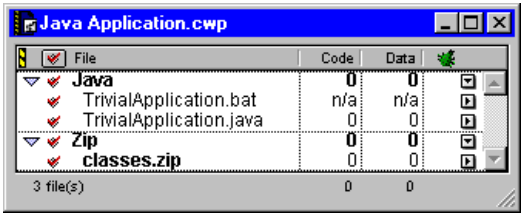
1. Create a new project. If CodeWarrior is not already running, launch CodeWarrior, and then choose the New Project item (Shift+Ctrl+N) from the File menu to create a new project. When the New Project dialog box appears, do two things:

1. Turn *off* the Create Folder check box

2. Select the Java Application stationery

When you click the OK button, the standard dialog box appears so you can save the new project. Name the project LifeApp.cwp, and save it inside the Start Code folder for this chapter, which is inside the Exer13 folder. When the new project window appears, notice that it includes two placeholder source files named TrivialApplication.bat and TrivialApplication.java (see Figure 13.7). You'll be replacing these files with new source code in a subsequent step.

FIGURE 13.7
A new Java application project ready to go



The presence of the .bat file is an indication of how CodeWarrior works with Java applications. A Java application must run under the control of some kind of runtime environment. This might be an interpreter or a just-in-time compiler, but either way some other bit of software executes and controls the Java application. CodeWarrior uses the Jview utility. The .bat file contains the instructions that launch Jview and in turn have it open and run the application.

2. Examine project settings. In this step you examine settings for the final output for the project.

Go to project settings, and examine the Java Project panel (Figure 13.2 earlier in this chapter). Notice that the type of project is already set to Application, and the output format is a Zip file named Java Lib.zip. All of these settings are acceptable. The final result from building this project will be a zip file named Java Lib.zip.

3. Add source files to the project. In this step you add the necessary source files to create the applet. You need to add three files, two in the Start Code folder for this project, and one in the Java Support folder inside the CodeWarrior folder.

To begin, go to the project window and select the Java group name (or one of the placeholder files in the group) so that new files will be added into this group. Then choose the Add Files item in the Project menu, and in the resulting dialog box navigate to the Exer13\Start Code directory. From this directory, add the following source files to the project:

- Life.bat
- Life.java

The application uses the same Life.java file as the applet project.

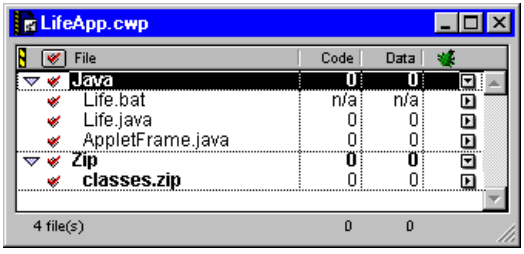
After clicking the Add button to add the files, repeat this process to add one more file. When the add file dialog box appears, navigate to the CodeWarrior\Java Support\Java Source directory. From this directory, add the following source file to the project:

- AppletFrame.java

After you have added this file, return to the project window. Select the two original placeholder files, and remove them from the project. Use the Remove Files option in the Project menu. When you're all through, the project window should look like Figure 13.8.

FIGURE 13.8

The Java application project with all the correct files in place.



4. Set the path to the zip file. The Jview utility must have a path to the zip file so that Jview can locate and launch the Java application archived in the zip file. If you examine the default TrivialApplication.bat file that is part of Java stationery, you'll see **%CWPROJPATH%Java Lib. zip** as the specified path, followed by the name of the class containing the **main()** function, in this case TrivialApplication.

The **%CWPROJPATH%** part of the path is a symbol that tells Jview to go to the folder containing the project. Inside that folder it will find the zip file. Recall that the default name for the output zip file is Java Lib.zip. All you need to do is change the name of the class containing the **main()** function, and Jview will find your zip file and your application. Right?

Right, as long as you're working on Windows NT. The **%CWPROJPATH%** symbol doesn't work right on Windows 95 at the time of this writing. This is a known bug in the CodeWarrior Java implementation, and may be fixed by the time you read this. However, to be on the safe side, I'm going to give you instructions for how to set the complete path name manually.

Here's how: Open the Life.bat file and examine the path to the zip file. Instead of the **%CWPROJPATH%** symbol, the initial text says:

C:\CodeWarrior\MCWProjects\Exer13\StartCode\Java Lib.zip

You must set this path to match the path to your project and zip file. I cannot tell you the precise text to type, because I don't know where you installed CodeWarrior or the exercise for this chapter.

WARNING You should note that no spaces occur in any of the directory names in this path. The presence of a space in a directory name terminally confuses Jview.

After you have the path set properly, save your changes and close the file. Closing the file is important, because Jview cannot read the file

if you have it open in CodeWarrior. Once the file is closed, you are ready to proceed to the next step.

5. Run the application. The project is ready to run, so choose Run from the Project menu. This tells CodeWarrior to compile, link, and launch the applet. CodeWarrior uses the first .bat file in the project to launch and run the application. In this case, that's Life.bat, which is designed to launch Jview and tell it to open and run the Life application.

A few moments after you issue the Run command, Jview appears, and the applet will be running. This looks mighty familiar—it's déjà vu all over again.

Building an application is just as easy as building an applet. Of course, CodeWarrior can't write your source code for you, but it does just about everything else. The CodeWarrior stationery provides you with everything you need, including the batch file to launch Jview. If you examine that file, you'll see that it specifies the name of the file in which to locate the application, in this case LifeApp.zip. This is the name you specified in step 2. You also specify the name of the class that is the starting point of the application, in this case Life.

In addition to applets and applications, a third kind of Java project exists—a library. You build a Java library exactly the same way you build an application. You can then add that library (typically a zip file) to your project in exactly the same way you would add classes.zip or any other file.

Summary

You have now encountered two completely and radically different targets—the x86 target and the Java Virtual Machine. It is remarkable how similar the development process is for both these targets. You can draw some important conclusions from this experience.

The CodeWarrior tool set is modular and reusable, as with any good code. The same tools—the editor, browser, project manager, and so forth—work for both targets. Because CodeWarrior is a multilanguage, multitarget, multihost environment, good ideas from different parts of the programming universe are incorporated into the tools, and help you program for a variety of targets.

In addition, your skills also become highly reusable because once you master the tools, the same knowledge applies to all targets. You'll see these benefits again in the next chapter, when you build a Mac OS project from scratch.

CHAPTER

FOURTEEN

14

Programming for Mac OS

- Setting Up for 68K Programming
- Setting Up for PowerPC Programming
- Creating a Mac OS Application
- Debugging Mac OS Code

If you have never programmed for Mac OS, then a new world awaits your discovery, one with millions of users (and potential customers). CodeWarrior's cross-platform capabilities can give you a head start on your journey. Whether the task at hand is porting an existing application or starting from scratch, you're looking at the right tools.

As with Windows, Mac OS is a fairly complicated target. Just for starters, remember that this operating system can run on two completely different families of processors, Motorola's 680x0 series and the PowerPC series—the 601, 603, and 604. These two chip families are as different from each other as they are from the 80x86 series, and programming specifically for one or the other has its idiosyncrasies. While the 68K can run emulated on a PowerPC Mac (and thus is the lowest common denominator for Mac programming), you can also build what is called a "fat" application that contains native code for both chips. CodeWarrior handles all of this with aplomb. If it runs on a Mac, you can write the code for it in CodeWarrior.

NOTE

Although you can use CodeWarrior on your Windows machine to create a Mac application, you still need a Mac OS computer to run the application! If you don't have access to one, you won't be able to see the end result of your work.

This chapter shows you how to use CodeWarrior to program for the Mac OS target. We'll discuss some of the unique settings available to you. Except for the discussion of project settings, this entire chapter is a hands-on exercise in how to use CodeWarrior for Mac OS development.

If this brief look intrigues you, and you decide it's time to take the cross-platform plunge, you'll need two things:

- A great set of tools—CodeWarrior can give you this on both platforms
- Knowledge of Mac OS programming

This chapter is not going to teach you how to program for Mac OS. You're going to run into some pretty obscure details even in this overview. If you are not already familiar with the intricacies of Mac OS you may feel overwhelmed, but don't worry about that. The Mac, like Windows, is a great platform with a tremendous amount of power and flexibility. Like the other platforms, programming for Mac OS is a topic worthy of a complete book. I know—I've written one called *Programming Starter Kit for Macintosh* (Hayden Books, 1995).

The Future of Mac OS

As I put the finishing touches on this book, Apple Computer, Inc. has announced the purchase of NeXT and its operating system to form the basis for the next generation OS for Macintosh computers. This will bring an entirely new target into the CodeWarrior family, and a new language—Objective C. Simultaneous with Apple's announcement, Metrowerks announced that it will provide Objective C compilers and linkers for the new operating system, ensuring a smooth transition for developers around the world.

I cannot predict what shape the new Mac OS will take, how it will be implemented, or what particular idiosyncracies you'll have to accommodate as you program for this target. However, the true value of CodeWarrior becomes immediately apparent.

Despite the radical changes in fundamental software technology represented by Apple's purchase of NeXT, the same set of tools will take you to the new target using a new language. The flexibility of the IR lets you weave through these obstacles like a Porsche on the autobahn.

Setting Up for 68K Programming

Because the Mac OS can run on two completely different chips, programming for Mac OS is, in a sense, programming for separate targets that happen to share a common operating system. Each of the chips—68K and PowerPC—has its own settings panels for project, code generation, and linker. In this section you'll examine the 68K settings panels and the options briefly. Then you'll look at the PowerPC panels.

After that, you'll build a simple Mac OS application. Along the way you'll see that setting up a project for either chip is pretty simple.

When you create a new project, the first thing you have to do is pick a target. We discussed how to do that back in the "Picking a Target" section of Chapter 3. You go to the project settings, click the Target tab, and select the Mac OS 68K Linker as your target.

When you do, the panels available to you for project settings change to reflect your new target. The panels specific to 68K development are:

- 68K Project
- 68K CodeGen
- 68K Linker

We'll look at each of these in some detail. Also note a disassembler for 68K code, and a panel for CFM68K—the code fragment manager version of 68K that supports shared libraries. If you're interested in these panels, you can find out about them in the CodeWarrior documentation, especially in the *Targeting Mac OS* manual.

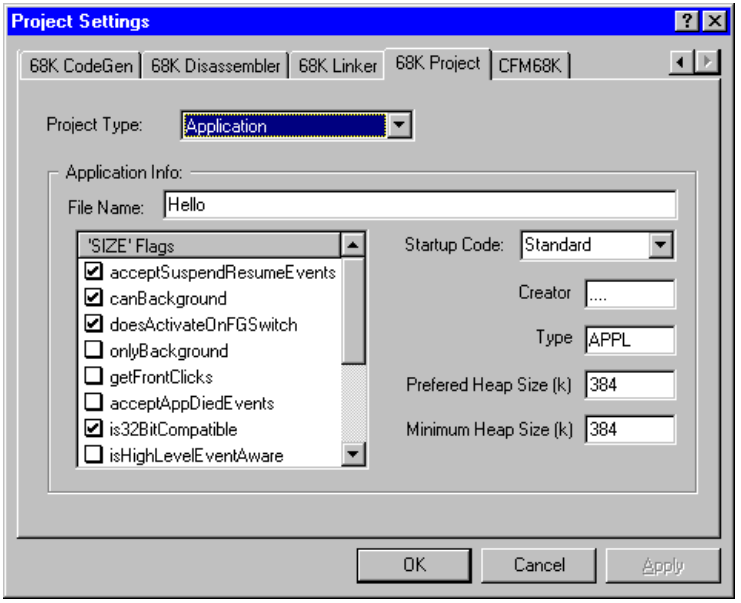
68K Project Settings

Figure 14.1 illustrates the 68K Project panel, and the options available to you. They are:

- **Project Type**—choose from application, code resource, static library, CFM68K application, CFM68K shared library, or MPW Tool

- **File Name**—specify the name of the final program
- **'SIZE' Flags**—settings that determine how Mac OS interacts with the application (stored in the SIZE resource in the application)
- **Startup Code**—initialization code you can choose to provide support for a Mac-hosted debugging tool called The Debugger
- **Creator**—the application's unique four-character creator code
- **Type**—for an application, the four-character symbol APPL
- **Preferred Heap Size**—the amount of memory to allocate to the application
- **Minimum Heap Size**—the minimum amount of memory required to run the application

FIGURE 14.1
Use the 68K Project settings to determine the type of project you are working on.



If you choose a project type other than application, the available options will change. Each of these projects has different requirements,

and because some may be unfamiliar to you, they deserve at least a little explanation.

A code resource is a self-contained chunk of code that you can ultimately add to an application. For example, menu appearance and behavior on the Mac are controlled by an MDEF (menu definition) code resource. You can write a code resource to create custom menu behavior, dialog box controls, window behavior, and so on.

A shared library is analogous to a DLL. The traditional 68K programming model does not support this type of project. However, CFM68K does. The "CFM" stands for Code Fragment Manager. On the Mac, a shared library is called a code fragment. Using CFM68K, you can create shared libraries.

Finally, an MPW tool is a utility that works under the auspices of Macintosh Programmer's Workshop, Apple Computer's own programming environment.

Each of these project types requires very different settings. I won't detail all the settings for all project types. *Targeting Mac OS* in the CodeWarrior documentation does that for you. I'll just stick with an application.

A project's file type and creator are used by the Mac OS to identify the kind of file, and the creating application. This is similar in function to the Windows registry where certain kinds of files are associated with particular applications.

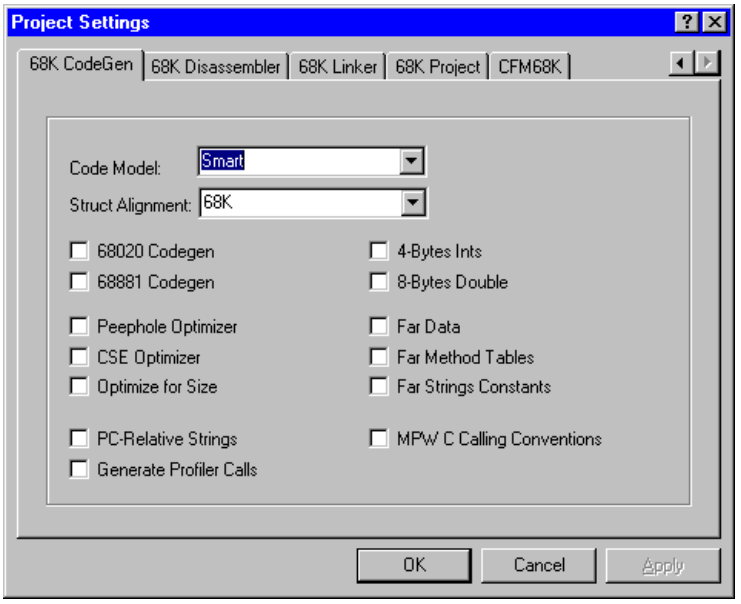
Clearly the Mac programming universe is rich and varied. However, you can see certain similarities between the kind of information required for a Mac application and that required for Windows. You set project type, name, and memory allocation values for both. And, as with the x86 Project settings, your best bet is to use stationery as the starting point for your project, in which case most of these settings are taken care of for you.

68K CodeGen Settings

This panel controls how the 68K back-end compiler works. Unlike the x86 model, where the IR is optimized, the 68K compiler implements optimizing in the back-end exclusively. You control optimizing and a few other options in this panel, as shown in Figure 14.2.

FIGURE 14.2

Use the 68K CodeGen settings to control the back-end compiler.



A bit of background about code segmentation will help you understand some of these options, and options in the 68K linker you'll meet in just a bit.

The 68K programming model allows programs to consist of several linked blocks of code (segments) that swap into and out of memory. Remember that the 68K programming model was developed more than a dozen years ago, when 128K of memory was all that was available in

top-of-the-line computers. A segmented-code model allowed large applications to work in small memory areas by swapping parts of the code in and out as necessary. An entire application is not necessarily memory-resident at all times.

With that in mind, let's turn to the individual options in this panel. Two important choices are:

- **Code Model**—choose from small, smart, or large
- **Struct Alignment**—choose from 68K, 4-byte 68K, or PowerPC

The small code model on 68K limits a single code segment to about 32K, because jumps within the segment are made forward or backward through a signed 16-bit value. The large model uses 32-bit values throughout. The smart model uses 16-bit values when it can, and 32-bit values when it must.

Struct alignment is a serious issue because of differences between the 68K and PowerPC chip. The choice of 68K is the lowest common denominator that works on all Macs regardless of chip.

Additional principal items in this panel control the following:

- **Specific Chips**—enable the compiler to generate instructions specific to the 68020 or 68881 chips (mostly math-related)
- **Optimizing**—choose among Peephole, Common SubExpression (CSE), and size
- **Data Sizes**—4 bytes for int values and 8 bytes for double values
- **Far Options**—another means (like the large code model) of circumventing the 68K chip's use of 16-bit values. Using various Far options lets you have more than 64K of data.

The data size options are a bit unusual, and might need a little explaining. The 68K programming model supports either 2 or 4 bytes for int values, and 8, 10, or 12 bytes for double values, depending on how you want to work with floating point values. The *Targeting Mac OS* manual covers all the details of how to specify the size of numbers for 68K.

You also use this panel to turn on profiler calls. At this time the profiler is a Mac-hosted tool only, but once you get your code over to a Mac, you could profile it there.

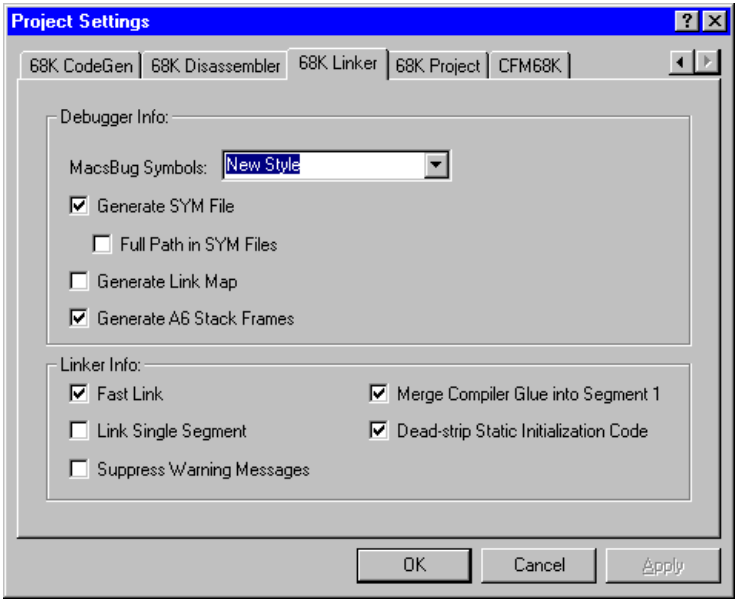
Once again, you can see that the options presented to you involve intricate and subtle details of the chip and the operating system. Default values will serve for typical projects. If you are familiar with the platform, these choices become fairly straightforward.

68K Linker Settings

The 68K linker panel, like the x86 panel, controls how your code is put together and how it is debugged. Figure 14.3 shows this panel, which is divided into two sections, one for debugging and one for linking.

FIGURE 14.3

You control debugging and linking in the 68K Linker panel.



The debugging options are:

- **MacsBug Symbols**—provide information for low-level debuggers
- **Generate SYM File**—create a SYM format symbolics file for debugging
- **Full Path in SYM Files**—use full path names in the SYM file
- **Generate Link Map**—create a link map
- **Generate A6 Stack Frames**—always create a stack frame even when not absolutely necessary

The debugging options should be pretty familiar to you, even if you've never programmed for the Mac OS. The only one that might need explaining is the last one about stack frames.

A6 is the 68K chip's register that is used as the stack pointer. For each function call where parameters are passed, the compiler builds a stack frame that enables a debugger to walk up the calling chain. If no parameters are specified, no stack frame is necessary, but you may want one anyway for debugging. To debug your code, you would turn this option on. You might turn it off when creating a finished product, however, because function calls that have no parameters are faster if this option is off.

The linker options in this panel range from the obvious to the truly obscure:

- **Fast Link**—trades memory for speed
- **Link Single Segment**—tells the linker to create one segment
- **Suppress Warning Messages**—turn off warnings (you still get errors)
- **Merge Compiler Glue Into Segment 1**—see the following explanation
- **Dead Strip Static Initialization Code**—remove such code that is unused

The Merge Compiler Glue option puts all compiler-generated code (such as inline functions that can't be inlined) into the first segment of a multisegment application. This prevents such code from being scattered into different segments, which would cause segments to be loaded at essentially unpredictable times. This can be a Bad Thing. On the other hand, because of segment size limits (effectively, 32K), if there is a lot of compiler-generated code, the first segment can grow too large.

As I said before, understanding the ramifications of some of these choices requires a deep knowledge of the foibles of how the OS works with the chip. Most of the time, default values for these obscure settings will do just fine, even when you're not 100 percent sure what they're doing. As proof, you're going to build a real Mac OS application while sitting at a Windows machine. First, however, let's take a look at the PowerPC chip.

Setting Up for PowerPC Programming

As I mentioned before, Mac OS runs on two completely different chips. The PowerPC is an extremely fast and powerful RISC (reduced instruction set) chip that is breaking speed records on a regular basis. The raw speed of this chip makes it very appealing for demanding computational tasks such as image processing, sound management, and so on. Production machines are currently available with multiple PowerPC processors, further enhancing the raw power of PowerPC-based machines.

As with all other targets, when you choose the PowerPC target for Mac OS, the available panels in the Project Settings dialog box change to reflect your choice. For PowerPC programming, the panels include:

- PPC Project
- PPC CodeGen
- PPC Linker

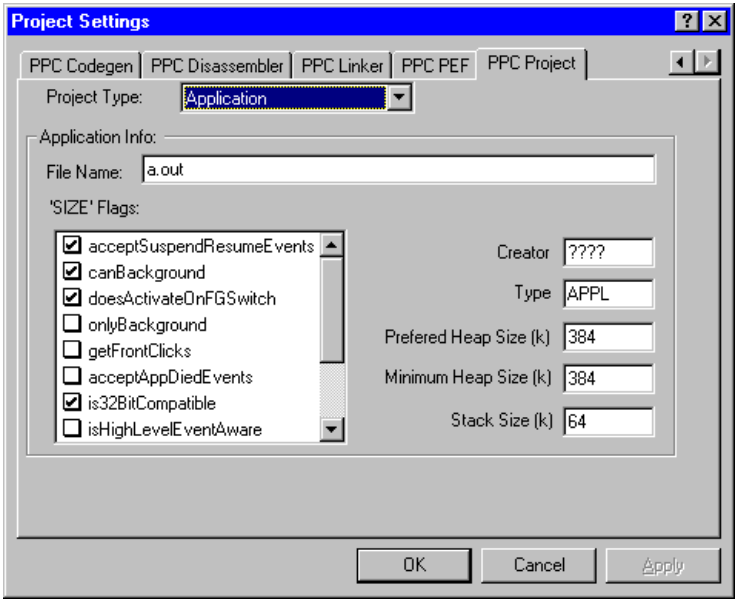
We'll look at each of these in some detail. Also note a disassembler for PowerPC code, and a panel for PPC PEF. PEF is an acronym for Preferred Executable Format, and is the data storage format for PowerPC code. If you're interested in these panels, you can find out about them in the CodeWarrior documentation, especially in the *Targeting Mac OS* manual.

PPC Project Settings

Figure 14.4 shows you the options available for a PowerPC project. All the options are identical to those in the 68K target, and you can refer to that discussion earlier in this chapter for details. One 68K option is missing, however: the Startup Code option. All PowerPC projects use the same startup code.

FIGURE 14.4

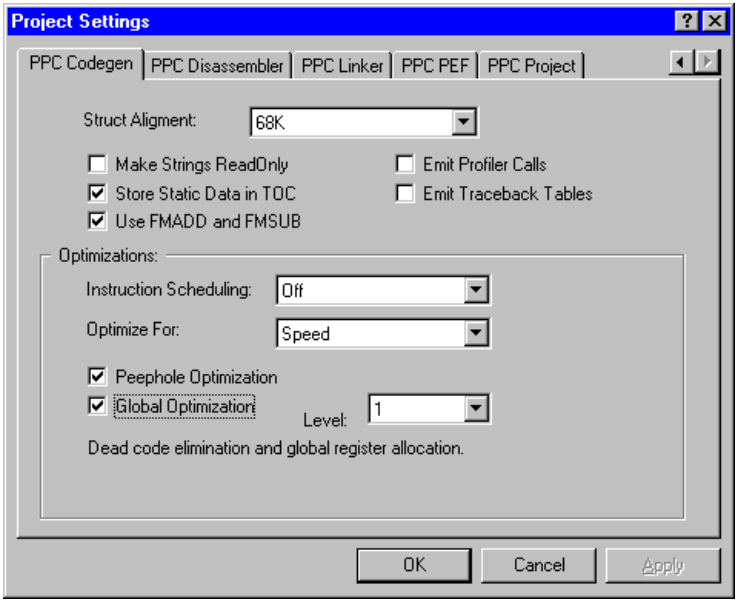
Use the PPC Project settings to determine the type of project you are working on.



PPC Codegen Settings

As with the other Codegen panels, this panel controls a specific back-end compiler. Figure 14.5 shows you the available options.

FIGURE 14.5
Use the PPC Codegen settings to control the back-end compiler.



This panel gives you the same Struct Alignment choices you had in the corresponding 68K panel. Aligning for the 68K chip is the lowest common denominator, and is useful if the data files the PowerPC program creates will be read by a 68K version of the same software. If your code is going to run purely on PowerPC machines, the PowerPC alignment will result in faster code.

In addition to Struct Alignment, notice a series of five check boxes that control various aspects of the compiler. For the most part, these are subtle and obscure issues for which default values serve just fine. You can find full details on all of these options in the *Targeting Mac OS* manual.

The rest of the panel is devoted to optimizations. As with the 68K compiler, the PowerPC compiler works in the back end, not through IR. The choices are:

- **Instruction Scheduling**—pipeline instructions for enhanced performance, tuned to a particular PowerPC chip
- **Optimize For**—choose speed or size
- **Peephole Optimization**—turn it on to activate this optimization strategy
- **Global Optimization**—optimize throughout the code at the chosen level
- **Level**—choose a level from one to four

Your first choice is Instruction Scheduling. The possibilities for this setting are Off, 601, 603, and 604. Each number refers to a particular member of the PowerPC chip family. This strategy rearranges instructions in such a way that they arrive at the chip in an order that maximizes throughput on the chip. Scheduling is best used when you know your code will run on one kind of chip only, because scheduling performed for one chip may actually impair performance on another.

The speed and size options have the same effect as on the 68K target.

When you choose a level for the global optimizer, the optimizing strategies for that level appear in the panel. Table 14.1 shows you what each level does. Each level adds new strategies to all those in lower levels, so level 4 performs all optimizations listed in the table.

TABLE 14.1 Optimizing strategies for the PowerPC back-end compiler

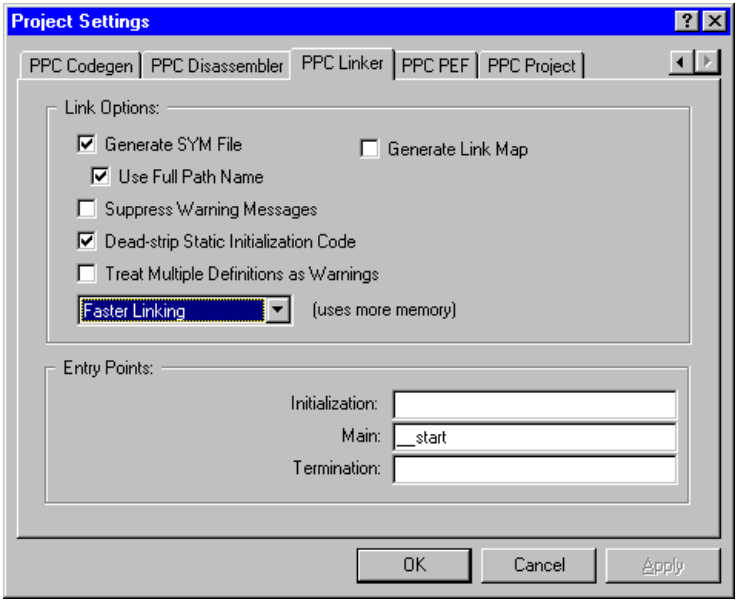
Level	Strategies
1	dead code elimination, global register allocation
2	common subexpression elimination, copy propagation
3	loop invariant code motion, strength reduction, loop transformations
4	repeated common subexpression elimination, repeated loop invariant code motion

PPC Linker Settings

The PowerPC (PPC) Linker settings panel, shown in Figure 14.6, is divided into two sections—one for linking, and one for entry points.

FIGURE 14.6

The PPC Linker panel controls how the code is put together.



The Link options in the PCC Linker section are similar to and have the same effect as those in the 68K Linker panel, and you should refer to that discussion earlier in this chapter if you have questions about these options. Once again, the speed selection trades speed for memory. The fast linking option requires more memory; the slower option takes twice as long but requires much less memory and is useful when working with big projects.

The Entry Points section of this panel has three fields:

- **Initialization**—the entry point for initializing code, called when the fragment is loaded into memory

- **Main**—the entry point for the main function, called when the fragment is executed (ignored for shared libraries)
- **Termination**—called when the fragment is purged from memory

The Main entry point is required for applications and is optional for all other project types. The other two entry points are optional for all projects—use them depending upon what your code does and needs.

Creating a Mac OS Application

In this exercise you are going to create a Mac OS 68K application from scratch. As always, because the point here is to learn the tools, I'll provide the code. Actually, Brad Mohr is the author of this particularly colorful "Hello, World" program. Brad is the senior analyst for the Technical Japanese graduate studies program at the University of Washington in Seattle, and is an accomplished Mac OS programmer. Even if Mac OS programming is new to you, you can successfully build a Mac application in this exercise.

The reason you're building a 68K application instead of a PowerPC application is simple. As I pointed out in a couple of places in this chapter, the 68K is the lowest common denominator for Mac OS. If it runs on a 68K machine, it will run on a PowerPC machine. I have no idea what kind of Mac you have access to, so using 68K ensures that this project will work for you.

If you have a PowerPC Mac, you can build a PowerPC project instead of 68K. The process is virtually identical. As you saw in the discussion of the settings for programming, significant areas of similarity exist in the setup work. In areas where the two chips differ, default values will serve you just fine.

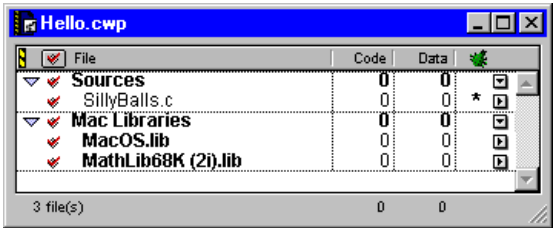
At the end of each step in this exercise, I'll point out where you might do things a little differently if you're building a PowerPC version of this application.

1. Launch CodeWarrior and create a new project. Launch CodeWarrior, and then choose the New Project item (Shift+Ctrl+N) from the File menu to create a new project. When the New Project dialog box appears, do two things:

1. Turn *off* the Create Folder check box
2. Select the Mac OS Basic Toolbox 68K application for C

When you click the OK button, the standard dialog box appears so you can save the new project. Name the project Hello.cwp, and save it inside the Start Code folder for this chapter, inside the Exer14 folder. When the new project window appears, as shown in Figure 14.7, notice that it includes a placeholder source file named SillyBalls.c. You'll be replacing this file with new source code in a subsequent step.

FIGURE 14.7
The new 68K application project is ready to go.



The other two files are libraries. MacOS.lib contains the core of the entire Mac OS. The MathLib68K(2i).lib file contains math routines using 2-byte integers, the default for 68K development.

NOTE If you're building a PowerPC project, use the PowerPC stationery. It will have a different collection of libraries.

2. Modify and examine project settings. In this step you modify some project settings for this particular application. This is a very simple appli-

cation that does not take advantage of some features of Mac OS. As a result, you need to change a couple of settings.

First, go to the 68K Project panel for this project. Change the file name to MCWHello. This is the name of the application you are about to produce. Next, turn off three of the SIZE flags, specifically:

- acceptSuspendResumeEvents
- canBackground
- doesActivateOnFGSwitch

This particular application does not know how to handle being sent to the background, so you're turning off support for background services in Mac OS. When you're all done, the only flag left checked should be:

- is32BitCompatible

Figure 14.8 shows this panel after you have completed all the changes.

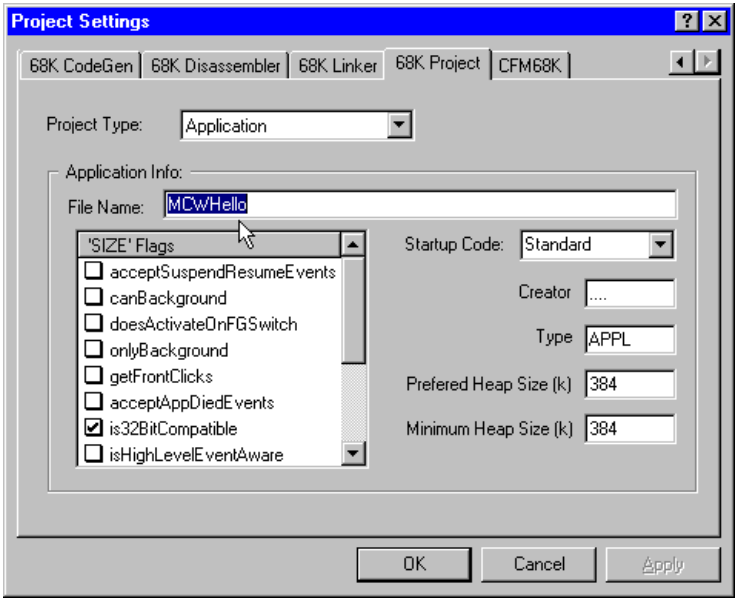
The default values for all other options in this panel are fine. Similarly, the default settings for 68K CodeGen and 68K Linker are also right for this application; that's the beauty of stationery. Take a look at those panels now, just to see what the settings are.

When you are through, click OK to save the changes you made to the 68K Project panel, and continue with the next step.

NOTE For PowerPC, use the PPC Project panel, but do exactly the same things.

3. Add source files to the project. In this step you add the necessary source files to create the application. To begin, select the Sources group name so new files will be added into the correct group. Then choose

FIGURE 14.8
The 68K Project settings prepared for the MCWHello application



the Add Files item in the Project menu, and in the resulting dialog box navigate to the Exer14\Start Code directory. From this directory, add the following source files to the project:

- Hello.c
- Hello.rsrc

Click the Add button to add the files to the project.

The Hello.c file contains the source code for this simple application. Feel free to examine the source file. It types out the words “Hello World” in a Mac window, complete with sound effects.

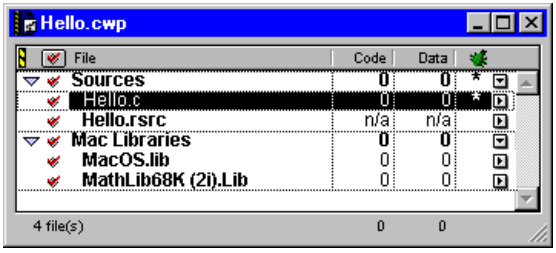
The Hello.rsrc file contains the sounds that play with the application. The 68K linker recognizes the .rsrc file extension as indicating a resource file, and merges the data with the application appropriately.

NOTE

Mac OS applications use many different kinds of resources. The Constructor rapid application development tool covered in Chapter 11 is a Windows-hosted resource editor for some fundamental resource types. For other kinds of resources, such as the sounds in this application, you really need a Mac OS computer to create and edit the resource information. No comprehensive Mac resource editor is available that runs under Windows 95/NT.

After adding the two files, return to the project window. Select the SillyBalls.c placeholder file, and remove it from the project. Use the Remove Files option in the Project menu. When you're all through, the project window should look like Figure 14.9.

FIGURE 14.9
The 68K application project with all the correct files in place



NOTE

For PowerPC development, use the same files! You'll find no difference in the code at all. When you stop to think about it, that's astonishing. The compiler takes care of all the work—the back-end code is generated appropriately for the different chips from the same source. You get two for the price of one. Good deal.

4. Build the Application. You can't run the application on your Windows machine, so choose Make from the Project menu to compile and link the final product. If you go to the Windows desktop and look inside the Start Code folder, you should find a file named MCWHello. This is the Mac OS application, just about ready to run.

While you're looking in the Start Code folder, notice that a SYM file lurking here as well. The default settings for the linker told the linker to create a SYM file. Although you're not going to do it in this exercise, you could bring that SYM file over to a Mac OS computer and debug the code you just wrote (you'd have to bring the source files also).

NOTE For PowerPC, the process is exactly the same.

5. Bring the Application to a Mac OS Computer. This can be a little tricky the first time you do it, so follow along closely.

Put a DOS-formatted floppy disk in your machine, and copy MCWHello to the top level of the floppy disk. You're not done yet, you're only halfway there. Read on!

Remember that CodeWarrior puts the resource fork of a file in the Resource.frk folder. The MCWHello application has a resource fork shadow, and it is vital that you move *both* the main file and the resource fork shadow to the Mac. Here's what to do to ensure everything works all right.

1. Create a folder on your floppy disk named Resource.frk.
2. Copy the MCWHello shadow file from the Resource.frk folder inside the Exer14\Start Code directory, and put it inside the new Resource.frk directory on your floppy disk. You have copied both of the MCWHello files to your floppy, and they remain in the same relationship they had on your hard drive—the shadow file is in a Resource.frk folder at the same level as the main file.
3. Now bring the floppy to your Mac OS computer (on which the Mac utility PC Exchange has been installed), and put the disk in the Mac floppy drive. The icon for the disk will appear on the Mac desktop.

4. Double-click the icon to open the disk, and copy MCWHello over to your Mac. You only need to move the main file. You cannot see the Resource.frk folder when you are on the Mac desktop, but the Mac OS knows about the folder and will copy both parts of the file properly.

NOTE For PowerPC, the process is exactly the same.

6. Set the file's type information. At this point you have successfully copied MCWHello over to your Mac, but you can't run it just yet. On the Mac you can only launch files of type APPL (remember the file type and creator information in the 68K or PPC Linker panel). Unfortunately, that information is lost in the file transfer. This means you have to set the correct file type after the file arrives on the Mac, so that Mac OS recognizes this file as an actual application.

Various utilities and resource editors give you access to a file's type information. For example, you can set this information using ResEdit, a free resource editor from Apple Computer (available for free download from Mac-oriented programming sites). Other tools are also available. Locate such a tool and modify the file's type to be APPL. Here's how you can do it with ResEdit.

Launch ResEdit and open the MCWHello file in ResEdit. Choose the Get Info For MCWHello item in the ResEdit File menu, and a dialog box appears. One of the fields in this dialog box is the file type, which you can change to APPL. Then just save the changes and close the file.

NOTE Again, for PowerPC, the process is exactly the same.

7. Run the application. Now that you have successfully copied the file to your Mac and changed the file type to APPL, just double-click the icon to launch the file. Then just sit back and enjoy the show. Figure 14.10 shows you what the window looks like. To end the program, click the mouse anywhere in the window.

FIGURE 14.10
The Hello, World program, running on a Mac.

NOTE For PowerPC, the process is exactly the same.

The application senses the bounds of the window and breaks the text when the next letter doesn't fit. While looking at the program (or at Figure 14.10) you might think that the lonesome letter "d" on the second line was a mistake. Actually, the text is large and wraps intentionally so that you can hear the carriage return sound built into the application. Feel free to modify the code to change the font size, or modify the lexical fireworks that appear right after the program finishes typing the text.

That's it for this exercise. Notice that except for starting with different stationery, everything about using CodeWarrior for either 68K or PowerPC is identical. In a full-blown project, you are likely to encounter some subtle differences, as pointed out in the sections that discussed the various options you have available to you for Mac OS programming on the two different chips. If you're writing code for one chip in particular you may want to take advantage of certain features available for that chip.

These issues are really related to programming for Mac OS and lie beyond this book's scope. The one thing missing here that we haven't discussed until now is how you debug your software while it's running on the Mac. Let's take care of that issue right now.

Debugging Mac OS Code

The full CodeWarrior for Windows 95/NT package includes a Mac-hosted debugger so you can debug a program while it runs on the Mac. You'll find the Mac-hosted debugger in a file titled Macutils.hqx on the CodeWarrior CD. Copy that file to your Mac OS computer, and then run a binhex conversion program (such as the freeware utility Stuffit Expander) to decompress it. The decompressed file becomes an installer application that you can use to install the Mac OS debugger (and related files) on your Mac.

TIP

The same installer also includes two outstanding Mac-hosted utilities for studying your program's performance. ZoneRanger tells you how you're using memory, and Profiler helps you analyze function-level performance with microsecond precision. These tools are fully detailed in the CodeWarrior documentation.

To debug the software, bring the SYM file created by the Windows-hosted tools over to your Mac, and open it with the Mac-hosted

debugger. You won't have any difficulty working with this new debugger because it is essentially identical to the debugger you read about in Chapter 8.

Mac OS programmers can perform the same magic in the opposite direction—write and debug Windows software while working on the Mac. You can, too. The Mac-hosted tool set currently includes a two-machine cross-debugger that runs on the Mac but debugs Windows software running remotely on a Windows computer. Work is also underway to create a Windows-hosted cross-debugger for software running remotely on a Mac! When that tool appears, programming for Mac OS from your Windows computer will be even easier. Not only will you be able to work from your Windows machine, you'll get one more great benefit. When your software crashes on the remote machine, your programming machine remains unscathed. When the cross-debugger saves you from electronic oblivion, I promise you'll be smiling.

Summary

You have accomplished a great deal, and if you have made it this far you are well on your way to becoming a code warrior. You have seen all the tools and the three principal targets. It's time to start wrapping things up.

Before we jump to the final chapter, stop for a moment to consider what you have just accomplished in this chapter. It is nothing short of remarkable.

You created a CodeWarrior project for Mac OS on your Windows machine. You used prepared source files for your Mac code, but you could have easily written your own while working on your Windows machine with the CodeWarrior editor and browser. You compiled and built the application while on your Windows system. You generated

symbolic information on the Windows machine that you could also migrate over to your Mac for debugging.

Finally, you ran the program on your Mac, and it worked! You can do almost all your Mac OS programming from the comfort of your Windows environment. This is extremely cool.

Best of all, you employed the same environment and the same tools you used to create a DLL, an MFC application, a Java applet, and a Java application. The bottom line here is really simple—pick a language, pick a target, pick a host. The choice is yours. What's more, the future holds even more choices. CodeWarrior's future is the topic of our final chapter.

CHAPTER 15

FIFTEEN

CodeWarrior Tomorrow

- The Path You Traveled
- Cross-Platform Development
- Where CodeWarrior Is Going

The adventure is almost complete—the hard part is over. Now it's time to relax and contemplate your surroundings—to look back at the course you have followed with a new understanding—now that you have reached the end of the voyage. And, more importantly, to look at the significance of where you are and what lies ahead.

The Path You Traveled

When you began this journey, you may have had little inkling of the architecture that lies behind the CodeWarrior tools. If so, I hope you have been pleasantly surprised by the design's elegance. By rising above the common plane of ordinary programming environments and inventing the Intermediate Representation—the all-important IR that lies at the heart of CodeWarrior—the Metrowerks engineers have created a remarkably flexible and powerful tool.

You have discovered that using just one tool set you can write code in C, C++, Pascal, or Java and target Windows, Mac OS, and the Java Virtual Machine. As you traveled through this book you met all of the tools and each of the principal libraries of code that come with CodeWarrior, including the following:

- Project manager
- Source code editor
- Browser
- Plug-in compilers and linkers
- Debugger
- Constructor

- MSL C
- MSL C++
- Pascal Library
- Microsoft Foundation Classes
- Java API
- PowerPlant

You have also seen how to use these tools and libraries to put together code for each individual target covered in this book.

The real significance of this accomplishment, however, does not lie in the fact that you have yet another environment in which you can program for the Windows operating system. Not hardly. CodeWarrior's truly unique feature is the multiplicity of targets it makes available to you. With these targets come the power and the goal of CodeWarrior—to program in your choice of language, for your choice of targets, on your choice of platforms.

Cross-Platform Development

The boundaries between various kinds of computers and operating systems are fast disappearing. Network protocols enable computers using different chips and operating systems to communicate with each other. The Java runtime architecture attempts to define a universal machine that can work on all computers. Work proceeds on cross-platform libraries for desktop development. The power of desktop computers continues to grow exponentially. The presence of multiple chips inside a single box adds an entirely new dimension to the mix—parallel processing.

As the electronic boundaries disappear, concepts of great value cross the old borders and present themselves to you. CodeWarrior, as it stands today, is a tool that you can use to take advantage of these

ideas. New targets are more easily available to you than ever before. For example, the Mac OS-hosted tools include a compiler for Windows development, and the Windows-hosted tools include compilers for Mac OS. You can migrate to an entirely new continent of computing without ever leaving your favorite neighborhood and it's familiar operating system.

It is in this venue—cross-platform development—that CodeWarrior has no equal. Keep in mind that many more targets are available for CodeWarrior than just Windows, Java, and Mac OS. CodeWarrior already supports Be OS from some platforms, and will soon support a variety of chips designed for embedded systems. The first of these embedded targets is already available—the PlayStation game console.

As you have discovered for Windows, Java, and Mac OS, each target has its idiosyncrasies, and you'll certainly need to understand the internal intricacies of whatever operating system or API exists for your target of choice. But the same powerful tools and intuitive techniques you use in CodeWarrior work equally well for all targets. The process of moving to a new target and gaining new markets is easier, because you don't have to leave the development platform or the environment where you are comfortable.

As CodeWarrior evolves (currently a new release appears three times a year), more targets and enhanced tools will appear.

Where CodeWarrior Is Going

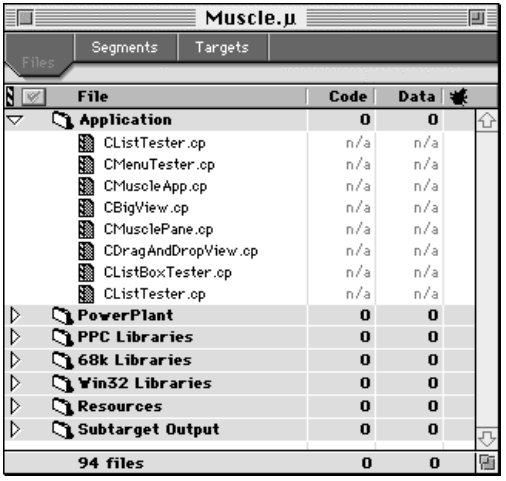
For clues on the direction that CodeWarrior will take tomorrow, look to the browser and the message window architecture we discussed at several places in Part III. Remember how they access data seamlessly and transparently, fetching the required information from whatever file it resides in without you having to open a file or even care which file contains that tidbit of information.

These are the newest features in the environment, and show you how the Metrowerks engineering team is answering the question, "What should a development environment be?" The answer is, it should be easy to use, friendly, intuitive, powerful, and smash the barriers between you and your code.

CodeWarrior's New Project Manager

Just over the horizon is a new project manager, and it's worth an advance peek. It epitomizes the long-range direction CodeWarrior is taking, and how Metrowerks brings powerful new features into the tool set without disrupting everything that has gone before. Figure 15.1 shows you what the new project manager window looks like running on Mac OS (as I write, it hasn't been implemented for Windows just yet).

FIGURE 15.1
The project window from an alpha version of CodeWarrior Version 2 demonstrates powerful new features.



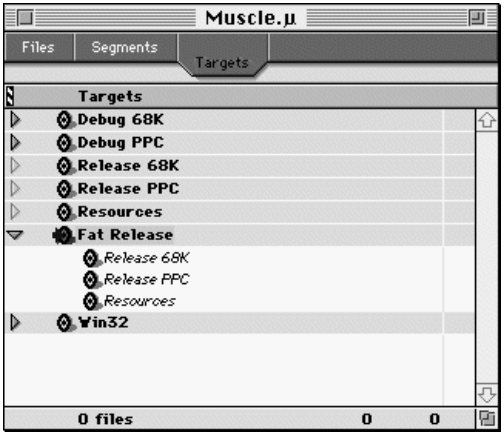
The choices represented in the bar across the top of the window—Files, Segments, and Targets—reflect a whole new dimension in

project management. Where the current CodeWarrior requires separate projects, the new CodeWarrior manages multiple targets in one project.

For example, you can have a single project that generates code for both Mac OS and Windows. Each target has its own settings, files, and libraries, but different targets can share common files. You can create debug targets where project settings are designed to create debugger symbolics or turn on debugging features you have built into your code. The possibilities are unlimited. Figure 15.2 shows you a list of (mostly) Mac OS-related targets for creating several different versions of the same application from within a single project.

FIGURE 15.2

A single project can contain multiple targets.



Different targets within a project can be dependent on other targets. For example, the Fat Release target highlighted in Figure 15.2 builds a Mac OS version of the project that includes native code for both the 68K and PowerPC chips. It requires that the 68K release, the PowerPC release, and the project resources be built first. The project manager knows this, and handles the dependencies automatically.

The same source files can be shared among multiple targets. All files for all targets are listed under the File view in the project window, as shown previously in Figure 15.1. You assign individual files to different targets as you wish. Using one project as an umbrella, you can cover all the different builds you want to create.

When you build a project, you can update one target or all targets. If a file is already compiled for one target, it doesn't have to be recompiled for another. If the compiler settings are the same for each target, the resulting object code can be reused again and again in different targets. However, if targets use different chips or the settings would result in different object code (for example, you choose different optimizations for different targets using the same chip), then the file is recompiled with the proper settings and you get the right object code for each target.

What the designers and engineers have accomplished here is to extend the functionality of the current project manager to a higher level. The same ease of use and flexibility is maintained. In the new project manager, issues such as cross-project dependencies and object code management remain completely transparent to you, just as they are now with cross-file dependencies. You have new power, new options, and new directions you can take, but you work with the manager just as you always have. The editor, the browser, project settings, and the other tools all work essentially the same, with some enhanced features appearing along with the new project manager.

This sneak preview of Version 2.0 cannot cover all the contemplated enhancements and improvements, but should give you a glimpse of what's coming and how CodeWarrior changes over time. The new project manager is a perfect example. It is a major step forward, and yet is incremental. As the tools evolve, the skills you have worked so hard to master remain 100 percent current. You don't have to learn how to use a new tool; instead, new power appears and your programming tasks become easier to manage.

The Electronic Revolution

It is fitting that this book end on a futuristic note, because you stand at the cusp of change. While this book has been about something eminently practical—how to use a particular set of tools to write code for a variety of computer platforms—a very exciting revolution is going on in the world, and CodeWarrior is a big part of it.

In the nineteenth century the world experienced an industrial revolution of immense magnitude. The effects it had on the globe were both pervasive and profound. The people who made that revolution happen were the inventors and tool makers—the small group of people who designed and built the machines that made the goods.

As we move into the twenty-first century we find ourselves in the midst of another worldwide transformation, the effects of which will certainly be as widespread (if not more so) as the industrial revolution of that earlier time. Who is the driving force behind this metamorphosis? That person is you, the computer programmer who writes the software to run on the computers that drive the electronic revolution.

You are the leading edge of a technological revolution sweeping the globe. But you yourself need powerful tools to get your work done—to build the virtual machines that will wing us all into the next millennium. This book has introduced you to tools you'll find useful and productive in your work. These pages have taken you a long way toward becoming a master of CodeWarrior. Our journey together is complete, but your adventure has only just begun. A universe of possibilities awaits you, code warrior.

APPENDIX A

Contacting Metrowerks

If you have purchased a full version of CodeWarrior, the *CodeWarrior QuickStart* manual has all the contact information you need for registration, updates, tech support, sales, and a list of distributors around the world. This appendix contains much of that same information for those who are using the CodeWarrior Lite demo version that comes with this book, *Mastering CodeWarrior for Windows 95/NT*.

Metrowerks Web Site

If you're Internet-connected, you have no better way to get into CodeWarrior than to surf to the Metrowerks Web site at:

http:\\www.metrowerks.com

You'll find an immense amount of information ranging from the latest updates to excellent T-shirts (part of the Geekware line). You can discover all kinds of things on the site—software updates, special deals on the tools, libraries of code, the best programming books, and a whole lot more.

Tech Support

Metrowerks operates a world-class tech support department for registered customers available at:

support@metrowerks.com

Tech support is free. You'll find forms for questions, suggestions, and bug reports in the Release Notes folder in the CodeWarrior directory on your hard drive. Use the forms, because then your question or suggestion will include all the information tech support needs.

You can also send messages to tech support from the Metrowerks Web site.

NOTE

Metrowerks does not provide tech support for CodeWarrior Lite, or for any book (including this one) that includes CodeWarrior Lite.

Sales

Metrowerks is extremely supportive of students interested in learning programming, and provides special academic pricing for both students and teachers. Site licenses are also available for both commercial and academic customers.

If you have questions related to purchasing CodeWarrior, customers in the United States can call the sales department at:

(800) 377-5416

Customers elsewhere can call the Metrowerks main office at:

(512) 873-4700

Customers all over the world can contact the sales folks by e-mail at:

sales@metrowerks.com

CodeWarrior is sold around the world. If you are a customer outside the United States or Canada, contact the sales department for the distributor in your sector of the planet.

Registration and Updates

Purchasers of CodeWarrior Gold who register with Metrowerks receive the next two CodeWarrior releases for free. Registering is simple. New purchases come with a registration card you can fill out and

mail. You can also register by using the Metrowerks Web site or by sending e-mail to the following address:

register@metrowerks.com

The Address

That industrious-looking fellow you see on the cover of the book busily pressing CDs full of code is hard at work at:

Metrowerks Corp.
2201 Donley Drive, Suite 310
Austin, TX 78758
USA

You can call Metrowerks at:

(512) 873-4700 (voice)
(512) 873-4900 (fax)

APPENDIX B

What's Inside CodeWarrior Gold

This book comes with a limited version of CodeWarrior known as CodeWarrior Lite. When you purchase and install CodeWarrior Gold (the full package), more than 200 megabytes of the latest and greatest tools shows up on your hard drive.

Perhaps best of all, if you purchase and register CodeWarrior you'll get two more free updates. As a result, every four months you get another release, and you stay right at the leading edge of software development technology.

So what's in all those millions of bytes of data? A lot! This appendix briefly describes the primary directories and their contents, as follows.

home.htm

This file, located at the root level of the CodeWarrior directory, is the gateway to a complete Web site installed on your hard drive as part of CodeWarrior. The Web site contains links to simple tutorials, the documentation, CodeWarrior books and their sample code, information on CodeWarrior products, and more. The CodeWarrior installer puts a shortcut to this file (called CodeWarrior Reference Web Site) in your Start menu so you have immediate and easy access to extensive information about CodeWarrior and Metrowerks.

bin

The bin directory includes the executable applications that are part of CodeWarrior, as well as miscellaneous support files.

The critical items in this directory are as follows:

- **Ide.exe**—the CodeWarrior integrated development environment (IDE)
- **MWDebug.exe**—the debugger
- **Constructor.exe**—Constructor
- **Plugins**—the directory holding all the compilers and linkers
- **Debugger Plugins**—the directory holding debugger plugins

You can come to this directory and launch the applications directly, or use the shortcuts installed in the Start menu. You should never have to deal directly with the contents of the two plugin folders.

BinSDK

The BinSDK directory includes a plethora of third-party utilities and related help files for Win32 programming. This directory also includes three subdirectories for 80386, Windows 95, and Windows NT utilities. A quick sampling of the many utilities includes:

- **Apfcvt.exe**—converts executables or libraries so that they load profiling libraries rather than their conventional libraries
- **APIMon.exe**—monitors a running application for all Application Programming Interface (API) calls
- **Capscrn.exe**—captures video from your monitor
- **DlgEdit.exe**—creates and edits dialog boxes
- **FontEdit.exe**—modifies existing fonts to create new fonts for your applications
- **Hw32.exe**—HeapWatch displays application heaps and monitors them for memory leaks
- **Stress.exe**—stress-tests applications in a low-resource environment

Many more tools and toys are provided in this directory for debugging, profiling, and managing your code.

CodeWarrior Documentation

This directory includes all the documentation in Acrobat (and occasionally HTML) format for CodeWarrior for Windows. For the initial release (DR1) there are 26 separate manuals organized along the same lines as the CodeWarrior architecture. These manuals include core documentation for the IDE, language documentation for the front-end compilers, targeting manuals for the back-end compilers, and library documentation.

The Documentation Overview file in this directory describes all the documentation and its organizational structure. You can link directly to this document (or to any individual manual) from the local CodeWarrior Web site.

The precise contents of this folder will vary over time as the documentation keeps pace with the tools.

(Help)

The (Help) directory includes help files for the IDE, debugger, and Constructor tools. The CodeWarrior Error Reference is also available as a help file. This document lists each CodeWarrior error and what you can do to correct the error.

To open a help file, go to the Help menu for the application, or navigate to the (Help) directory and double-click the icon for the file you want to see.

(Stationery)

The (Stationery) directory contains all the prebuilt stationery files for Win32, Java, and Mac OS development. You can add your own stationery as you see fit.

CodeWarrior Books

This directory contains the text for several works by nationally known programming authors, including the complete text of several published books in electronic form. These books can teach you a programming language such as C, C++, or Pascal, or how to program for Windows.

CodeWarrior Examples

This directory contains megabytes of example code covering all aspects of programming for Win32, MFC, Java, and Mac OS.

CW Release Notes

The CW Release Notes folder holds the latest, up-to-the-minute information about the CodeWarrior tools, including any last-minute changes, known bugs, and undocumented features.

Java Support

The Java Support folder contains all the libraries and source files necessary for Java programming. This includes all the Java packages from Sun MicroSystems as well as the Java Generic Library of container classes from ObjectSpace, Inc.

Mac OS Support

This directory contains all the libraries and source files necessary when you target Mac OS. The files are organized into the following subdirectories:

- **Headers**—the latest universal headers for the Mac Toolbox, and headers for various additional Mac OS components
- **Libraries**—compiled libraries for all Mac OS components, the Mac runtime libraries, profiler libraries, and more for both the 68K and PowerPC Mac targets
- **MacHeaders**—precompiled headers for Mac OS programming
- **PowerPlant**—all the source files and precompiled headers for the PowerPlant application framework

Metrowerks Standard Library

Within this folder, the MSL C and MSL C++ each has its own subdirectory. In each subdirectory you'll find the source files, header files, and compiled libraries (as well as the projects that build the libraries).

Resources

The Resources directory contains files necessary for the local CodeWarrior Web site installed as part of CodeWarrior.

Tools

The Tools directory contains a single file that, when brought over to a Mac, decompresses into an installer for the Mac-hosted versions of MW Debug, Profiler, and ZoneRanger. The debugger uses a SYM file generated on a Windows computer. Profiler is the viewing application

for data generated by the Metrowerks profiler libraries. The Metrowerks profiler is an active profiler that gives you microsecond timing accuracy for each function you profile. ZoneRanger lets you examine the application's heap zone for memory usage and leaks.

Win32-x86 Support

This folder contains all the libraries and source files necessary when you target Windows 95/NT. The files are organized into the following subdirectories:

- **(MFC4.2)**—the source files and the .pch file for the Microsoft Foundation Classes
- **Headers**—the header files for the MFC and Win32 SDK, as well as precompiled headers for both MFC and Win32
- **Libraries**—compiled libraries for Win32, MFC, runtime support, and Pascal
- **Interfaces**—the Pascal interfaces for the Pascal library

With each release, the material provided on CodeWarrior Gold will expand. New and enhanced tools will appear along with new and revised manuals, new libraries, and perhaps new languages and targets each with their associated files and support directories. As CodeWarrior evolves, registered owners of CodeWarrior Gold automatically stay current with the latest and greatest.

INDEX

Note to the Reader: Throughout this index page numbers in **bold** indicate primary discussions of a topic. Page numbers in *italic* indicate illustrations.

Symbols

& (ampersands) in regular expressions, **198–199**
> (angle brackets) for included files, 83, 297
(asterisks) in regular expressions, **194–195**
\ (backslashes) in regular expressions, **197–198**
{} (braces)
balancing, **129–131**, 150–151, *151*
for folders, 82
[] (brackets)
balancing, **129–131**
in regular expressions, **196**
^ (carets) in regular expressions, **196–197**
- (dashes)
for breakpoints, 359, 398
in regular expressions, **196**
in Source Browser window, 342–343
$ (dollar signs) in regular expressions, **197–198**
# (number signs) for preprocessor statements, 299
() parentheses
balancing, **129–131**
for folders, 83, 87, 297
in regular expressions, **195–196**
. (periods) in regular expressions, **194**
+ (plus signs) in regular expressions, **195**
" (quotation marks) for included files, 83, 297
; (semicolons)
compiler warnings for, 309
errors involving, 316–317

A

A6 register, 564
absolute access paths, 296
Abstract Windowing Toolkit (AWT), 17, 440
academic pricing, 595
accelerators, 4
acceptSuspendResumeEvents flag, 572
Access Paths panel, **81–84**, *82*, 178, *178*, 293–296, *295*
Acrobat Reader, 29
Activate Browser option, 224, 262–263, *263*
Add Files command, 64, 94, 321, 417, 429, 457–458
Add Files dialog box, 64, *64*, 94
Add marker command, 132, 154, *154*
Add Window command, 63, 93
algorithms, C++ library for, 421
aligning
bytes, 517
layout panes, **483–485**, *483–484*
structures, 562, 567
Alignment Palette command, 483
Alt key for copying text, 147
ampersands (&) in regular expressions, **198–199**
angle brackets (<>) for included files, 83, 297
ANSI C standard, 291, 414
Ansicppx86.lib file, 421–422
AnsiCx86.cwp file, 89
Ansicx86.lib file, 94, 417, 429
Apfcvt.exe utility, 601
APIMon.exe utility, 601

APPL file type, 576
AppletFrame.java file, 545, 550
applets, **543–544**, *544*
  classes for, 447
  classes.zip file for, **546–547**, *546*
  project settings for, 544
  running, **547**, *548*
  source files for, **544–545**, *545*
application frameworks, **17**, **440**
  exercises for, **456–460**
  fundamentals of, **17–19**, **441–442**
  Java, **447–451**, *448–449*, *451*
  MFC, **443–446**, *443*, *446*
  PowerPlant, **451–456**, *454*
Application Programming Interface (API), 12
  for Java Virtual Machine, 447
  plugin, 280
applications
  building, **530–531**, **574–575**
  file type for, **576**
  Java programming for, **548–552**, *549–550*
  Mac OS programming for, **570–578**, *571*, *573–574*, *577*
  moving to MAC OS computers, **575–576**
  paths for, **551–552**
  project settings for, **528–529**, 549, **571–572**
  running, **530–531**, **552**, **577**, *577*
  source files for, **529–530**, *530*, **550**, *550*, **572–574**, *573*
  Windows programming for, **528–531**, *529–530*
APPMODUL.CPP file, 531
Arrange Objects command, 484
Arrange Objects dialog box, 484–486, *484*
arranging
  layout panes, **481–486**, *482–484*
  windows, **137–138**, 155–156
Array window, 377, *377*, 381
arrays
  frameworks for, 18
  in memory, 381, *381*
  Pascal library for, 424
  variables for, **376–377**, *376–377*, 402–403, *403*
arrow keys
  for insertion point, 114–115
  in selecting text, 117
arrow pointers in searches, 180, *181*
assembly language, viewing, **382–383**, *383*
Assembly Language Reference, 523
assignments, compiler warnings for, 308
asterisks (*) in regular expressions, **194–195**
Auto Indent option, 123
automatic file saving, **139–140**
Automatically Launch Applications When SYM File Opened option, 349

B

back-end compiler tasks, 21
background color
  for pixels, 495
  for text, 126, 150
backslashes (\) in regular expressions, **197–198**
backward searches, 165, 172–174, 201
Balance While Typing option, 130
balancing punctuation, **129–131**, 150–151, *151*
Base Address option, 516
base classes
  in multi-class window, 228–229
  in single-class window, 239
Batch option, 186
batch reports, **185–186**, 207–210, *208–210*
  activating, **186–187**, *186*
  Search Results window with, **187–191**, *189*, 208–209, *209*
beeping
  in balancing punctuation, 130
  in searches, 172
bin directory, **600–601**
binary files, deleting, 100, 288
binhex conversion program, 578
BinSDK directory, **601–602**
bit manipulation, Pascal library for, 424
bitmaps, **493–495**, *493*
  color and patterns in, **497–498**
  editing pixels in, **497**
  sample views for, **496–497**

tools for, **495–496**
transparency in, **499–500**, *499*
black triangles in PowerPlant, 454
bold print in Project window, 55
books, CodeWarrior, 603
braces ({})
balancing, **129–131**, 150–151, *151*
for folders, 82
brackets ([])
balancing, **129–131**
in regular expressions, **196**
brackets (<>) for included files, 83, 297
Break on C++ Exceptions command, 366–367
Break on Java Exceptions command, 366–367
breakpoints
conditional, 360, **365–366**, 399–401, *400–401*
for exceptions, **366–367**
fundamentals of, **360**
for libraries, 350, 432, *433*
as markers, 365
missing, **362–363**
in Program window, 345
setting and clearing, **359–361**, *360*, 394, *395*, 398–401, *399–401*
in Source Browser window, 342–343, 357
temporary, **360–361**, 394, *395*
viewing and controlling, **364–365**
Breakpoints window, 364–365, *364*, 399–400, *400*
Bring Up To Date command, 98, 155, 262–263, 288, 315
browser, **216–224**
activating, **224–225**, *225*, 262–263, *263*
background of, **221–222**
catalog view in, **222–228**, *226*, 263–265, *264–265*
class declarations in, **255–256**
deep and wide disclosures in, **233–234**, *234*
exercises for, **262–274**
functions in, **256–257**
graphical view in, **228–239**, *229–232*, *234–237*, *239*
for Java, 451, *451*
for MFC, 445, *446*
for multi-class browsing, **239–249**, *240*, *242–244*, *246–248*, 268–270, *268–270*
for multi-class hierarchies, **228–238**, *229–230*, *234–237*, 265–266, *266*
multiple inheritance in, **234–235**, *235–236*
navigating code in, **254–257**
reviewing history in, **258–259**, *259*, 273–274, *274*
settings for, **260–262**, *260–261*
showing and hiding classes in, **230–232**, *231–232*
for single-class browsing, **249–252**, *251*, 270–272, *271–272*
for single-class hierarchies, 236, *237*, **238–239**, *239*, 266–268, *267*
Symbol window in, **252–254**, *253*, 272–273, *273*
views in, **222–224**, *223*, **254–255**
Browser Coloring panel, 260, *260*
Build Extras panel, **79–80**, *81*, 224–225, *225*
Byte Alignment option, 517
byte manipulation, Pascal library for, 424
bytecodes
.class files for, 542, **546–547**
in disassembling, 298
from Java compiler, 22, 538–539

C

C, libraries for, 16, **414–419**, *417*
C++, libraries for, 16, **419–423**, *421*
C/C++ Compiler panel, 92, 290–292, *291*
C/C++ Warnings panel, 308–309, *309*
.c extension, 286
.c++ extension, 286
C String format command, 373, 402
C strings for variable values, 372
calling chains
examining, 350
fundamentals of, **344**
walking through, **355–356**, *356*, 395, *396*
canBackground flag, 572
Capscrn.exe utility, 601
capture video utility, 601
carets (^) in regular expressions, **196–197**
case-sensitivity in searches, 170, 357
.cat extension, 225

catalog view, 218–219, **222–228**, *226*, 263–265, *264–265*
for class declarations, 255
for functions, 256
.cc extension, 286
CDessert class, 262
central classes, 238–239, *239*
CFM68K Code Fragment Manager, 560
ChangeMe.c file, 89, 93, 99–100
ChangeMe.h file, 89, 99–100
changes, undoing, **120–122**, 146
character constants for variable values, 372
Character format command, 373
characters
C library functions for, 416
in regular expressions, **194**, 196–197
for variable display, 373
check marks
in editor, 139, *139*, 144, *145*
in Project window, 55–58, 70–71, 286, *287*
Check Syntax command, 289
choosing vs. selecting, 4
circles
for breakpoints, 359, 361
drawing, 496
class browsing, **10–11**, *11*, 219, 222–224, **239–242**, *240*, *242*, 268–270, *268–270*
for class declarations, 256
Classes pane in, **242–244**, *243–244*
Data Members pane in, **246**, *247*
for functions, 257
Functions pane in, **244–246**, *246*
Source pane in, **241–242**, *242*, **247–248**, *248*
for viewing and editing code, **248–249**
class declarations in browser, **255–256**
.class extension, 286, 542
classes
browsing. *See* class browsing; single-class browsing
deep and wide disclosures in, **233–234**, *234*
hierarchy of. *See* multi-class hierarchy view; single-class hierarchy view
importing, **450–451**, *451*
multiple inheritance in, **234–235**, *235–236*
in object-oriented code, 217–218
showing and hiding, **230–232**, *231–232*
Classes pane, **242–244**, *243–244*
classes.zip file, **448–450**, **546–547**, *546*
Clear command
for layout panes, 480
for menus, 489
Clear All Breakpoints command, 359
Clear Current Watchpoint command, 386
Clear Toolbar command, 39
Clear Watchpoint command, 386
clearing
breakpoints, 359, 364
keyword lists, 127
layout panes, 480
menus, 489
toolbar, 35, 39
watchpoints, 386–387
Close command, 60, 142
Close All command, 142
closing
files in editor, **142**
groups, 65
projects, **60–61**, 100
CObject class, 452
code
in browser, **248–249**, *250*, **254–257**
compiling and linking. *See* compilers and compiling; linkers and linking
converting between platforms, 134
debugging. *See* debugger and debugging
disassembling, **297–299**, *298*
finding, **357–358**, *357*
navigating through, **355–358**, *356–357*
object-oriented. *See* object-oriented code
reusing. *See* libraries
running, **348–350**, *349*
skipping and repeating, **353–355**, *354*
stepping through, **347–355**, *347*, *353–354*
viewing and editing, **248–249**, *250*
Code column, 55, 288

Code Fragment Managers, 560
Code Model option, 562
code resources, 560
CodeView format, 329
  setting for, 334
  with Visual C++, 337
  for Windows programming, 518
CodeView Format option, 518
CodeWarrior Books directory, 603
CodeWarrior C/C++/Asm Language Reference, 292, 309
CodeWarrior Documentation directory, **602**
CodeWarrior Error Reference, 307
CodeWarrior Examples directory, 603
CodeWarrior Lite demo version, 88, 594
CodeWarrior Lite Setup application, **28–31**, *29–31*
CodeWarrior Pascal Language Reference, 292, 309
CodeWarrior QuickStart manual, 594
CodeWarrior Reference Web Site, 600
COFF object format, 522
Collapse All command, 369
color
  for bitmaps, **497–498**
  for browser, 260, *260*
  for pixels, 495–496
  for text, **126–129**, *127–129*, 148, 150, 500
color swatch, 126
color tool, 495
coloring
  register, **370–371**
  syntax, **126–129**, *127–129*, 148
Colors menu, 498
Command File option, 522
command IDs for menu items, 487, 490, *491*, 507
comments, color for, 126
Committed Heap Size option, 516
Committed Stack Size option, 516
Common SubExpressions option, 520
comparisons, compiler warnings for, 309
compatibility for Windows programming, 523
Compile command, 288, 314
Compiler column, 283
Compiler pop-up menu, 283
compiler relative access paths, 296
compilers and compiling
  architecture of, **21–24**, *21*
  check marks for, 55–56, 70–71, 286, *287*
  choosing, **278–286**, *279*, *281*, *283*
  code generation control in, **289–290**
  commands for, **288–289**, 314–315
  disassembling code, **297–299**, *298*
  errors and warnings with, **307–314**, *309–311*
  exercises for, **313–323**
  extensions for, 78, 283–286
  fundamentals of, **47**
  headers, **300–303**, *301–302*
  just-in-time, 331, 538–539
  language, **282–286**, *283*, **290–293**, *291*, *293*
  plugin, **279–280**, *279*
  preprocessors for, **299–300**, *300*
  resource, **522**
  selecting files for, **286–287**, *287*
  speeding up, **293–297**, *294–296*
  target, **280–282**, *281*
Condition column, 366
conditional breakpoints, 360, **365–366**, 399–401, *400–401*
Confirm "Kill Process" When Closing or Quitting option, 351
Console User Interface (CUI), 92, *92*
constants
  in browser, 218, 226–227
  for variable values, 372
Constructor, **464**
  background of, **464–465**
  for bitmaps, **493–500**, *493*, *499*
  exercises for, **502–508**
  help files for, 602
  launching, 502–503
  for layouts, **476–486**, *478–480*, *482–484*
  for menus, **486–492**, *487–489*, *491*, 506–508, *507*
  operation of, **465–467**, *466*
  resource management in, **469–476**, *470–471*, *474*
  for text traits, **500–501**, *501*, 503–504, *504*
  working with, **467–468**, *468*
Constructor.exe file, 601
Constructor Manual, 466, 480

constructors, debugging, 349
containers, C++ library for, 420
containment in layouts, **477–479**, *478–479*
context menus
    in class hierarchy view, 237–238, *237*, 266–267, *267*
    in Classes pane, 269, *269–270*
    in Functions pane, 245–246, *246*
    in Members pane, 246
    settings for, 261
    in Source pane, 244, 248
    for Symbol window, 252, 272, *272*
Context Popup delay option, 261
Control menu commands
    Break on C++ Exceptions, 366–367
    Break on Java Exceptions, 366–367
    Clear All Breakpoints, 359
    Kill, 348, 351
    Run, 343, 348, 350
    Step Into, 348, 352–353, 394
    Step Out, 348, 353
    Step Over, 348, 352, 393
    Stop, 348, 350–351
converting code between platforms, 134
Copy/Const Propagation option, 520
Copy to Expression command, 375, 403
copying
    in Constructor, 468
    pixels, 497
    project files, 60
    resources, 473
    text, 147–148, *147*
Courier New font, 123
.cp extension, 286, 526
.cpp extension, 286
CProduce class, 262
crashes from user settings
    of current statement arrow, 354
    of memory attributes, 474
    of register values, 382
    of variable values, 369
Create Folder option, 53–54, 88
Create New Resource dialog box, 469–471, *470–471*, 504, *504*
Creator option, 559
cross-platform programming, **585–586**
    debugger for, **330–332**
    stationery for, 51–52
Ctrl key
    in file selection, 62
    for insertion point, 114–115
    in text selection, 117–118
ctype.h file, 416
curly braces ({})
    balancing, **129–131**, 150–151, *151*
    for folders, 82
current statement arrow, 345, **353–355**, *354*, 393, *394*
Custom Keywords panel, 128–129, *128*
Cut command
    for layout panes, 480
    for menus, 489
    for resources, 473
    for text, 120
CW Release Notes folder, 603
.cwp extension, 53–54
%CWPROJPATH% symbol, 551
cycling through windows, 138, 155

D

dashes (-)
    for breakpoints, 359, 398
    in regular expressions, **196**
    in Source Browser window, 342–343
Data column, 55, 288
data forks, 91
data hiding, **219–220**
Data Members pane
    in multi-class browser, **246**, *247*
    in single-class browser, **250–251**, *251*
Data menu commands
    Clear Current Watchpoint, 386
    Clear Watchpoint, 386
    Collapse All, 369

Copy to Expression, 375, 403
for data types, 373
Open Array Window, 377
Open Variable Window, 374
Set Watchpoint, 385
Show Types, 372, 402
View Memory, 380, 385
View Memory As, 381
data size options, 562
data types, **372–374**, *373*, 401–403, *402–403*
.dbg extension, 378–379
Dead Code Elimination option, 520
dead code stripping
for Mac OS programming, 564, 568
missing breakpoints from, 362–363
for Windows programming, 520
Dead Store Elimination option, 520
Dead Strip Static Initialization Code option, 564
Debug column, *55*, 57, *72–73*, *72*, 333, *334*
Debug command, 288, 304–305, 336–337, 432
debugger and debugging
activating, **329**
for assembly language, **382–383**, *383*
background of, **330–332**
breakpoints in, **359–367**, *360*, *364*, 394, *395*, 398–401, *399–401*
exercises for, **389–405**
help files for, 602
killing programs in, **351**
launching, **336–337**, *338*, 391, *392*
libraries, 350, **384**, 432–435, *433–435*
log window in, **346**, *346*, 391
low-level features in, **380–383**, *381–383*
Mac OS programming, **563–564**, **578–579**
for memory, **380–381**, *381*
for multiple languages, **387**
for multiple projects, **388–389**
for multitasking, **387–388**, *388*
navigating through code in, **355–358**, *356–357*
operation of, **328–329**
Program window in, **343–345**, *343*, 391, *392*
for registers, **382**, *382*
running code in, **348–350**, *349*
saving debug information for, **378–379**
setting up, **332–336**, *333–335*, 390
skipping and repeating code in, **353–355**, *354*
Source Browser window in, **340–343**, *340*
stepping into routines in, **352–353**, 394–395, *396*
stepping out of routines in, **353**
stepping over routines in, **352**, 393
stepping through code in, **347–348**, *347*, 353, *353*
stopping program in, **350–351**
switching with source code, **358–359**
symbol information for, **71–73**, *72*
viewing and changing data in, **367–377**, *368*, *373–375*, *377–378*, 397–398, *398*, 403–404, *404*
watchpoints in, **384–387**, *386–387*
windows for, **339–340**
Windows programming, **531–534**, *532–533*, *535*
Debugger Plugins directory, 601
decimal digits for variable values, 371
declarations
in browser, **255–256**
defined, 5
vs. definitions, 227
in Source pane, 244
deep disclosures in browser, **233–234**, *234*
Default format command, 373
default options
for projects, **75–76**
stationery for, 52
Default Size for Unbounded Arrays option, 377
#define statements, 299
definitions
vs. declarations, 227
defined, 5
deleting
binary files, 100, 288
breakpoints, 359, 364
compiler list extensions, 284
Expression window variables, 376
file sets, **184–185**, *184*
files and groups, **66–67**
files from searches, **180**

keywords, 127
layout panes, 480
markers, 133
menu items, 491
menus, 489
pixels, 497
resources, 473
with search and replace, 161
text, **120**
watchpoints, 386–387
dependency lists, 80
diagnostics, C++ library for, 420
diagonal lines in browser window, 229, *230*
dialog boxes, utility for, 601
direction in search and replace, 162, 165, 201
disabled commands, 34–35
Disassemble command, 298
disassembling code, **297–299**, *298*
disclosure triangles
for classes, 230–233, *231*, *234*, 265–266, *266*
for Constructor, 467, 477
in Errors & Warnings window, 312
for group names, 55, 65
for layouts, 477
for multi-file searches, 176
in Program window, 345, 347, 368, 402
in single-class hierarchy view, 239
for stationery, 51
for structures, 368–369
Display Classes window, 471–472, *471*, 502
distributing layout panes, **485–486**
DlgEdit.exe utility, 601
DLLs. *See* dynamically linked libraries (DLLs)
documentation
for compilers, 292
directory for, **602**
for libraries, 415
for PowerPlant, 456
for projects, 48
Documentation Overview file, 602
doesActivateOnFGSwitch flag, 572
dollar signs ($) in regular expressions, **197–198**
Dow, Gregory, 452–453
drag-and-drop
for bitmaps, 497
in Constructor, 467–468, 473, 479
for groups, 65
for menus, 489
for moving text, 118–119, *118*, 147–148, *147*
in panes, 125
Drag and Drop Editing option, 119
drawing mode for text, 500
dropper tool, 494, 496
dummy files, 85
Duplicate Resource command, 473
dynamically linked libraries (DLLs), **57**
building, **527–528**
debugging, **384**
exporting functions for, **526–527**, *527*
project settings for, **524–525**
source files for, **525–526**, *526*
Windows programming for, **523–528**, *524*

E

Eclipse.cp file, 525–526
Eclipse.cwp file, 428–436, *430–435*, 528
Eclipse.dll file, 527, 533
Eclipse.dll.lib file, 527, 529
Eclipse.rc file, 529
EclipseApp.cp file, 529–530, 532
EclipseDLL.cwp file, 524, *524*
Edit menu commands, 33
Clear, 480, 489
Cut, 120, 473, 480, 489
Duplicate Resource, 473
Edit Resource, 473
Find, 357
Grid, 483
Menu Info, 489
Menu Item Info, 491
for moving text, 118
New Menu, 488, 507
New Menu Item, 490, 507

New Resource, 469, *470*, 503
New Separator Item, 490
Preferences, 38, 74, 119
Project Settings, 76, 89, 127
Redo, 120–121, 146
Resource, 473
Resource Info, 474
Shift Left, 123, 152
Shift Right, 123, 152
Undo, 120–121, 146
editing
bitmaps, **493–500**, *493*, *499*
code, **248–249**, *250*
in Constructor, 468, *468*
keyword lists, **126–129**, *128*
layouts, **476–486**, *478–480*, *482–484*
libraries, 427
memory attributes for resources, 474
in Memory window, 380
menus, **486–492**, *487–489*, *491*, 506–508, *507*
object properties, 505–506, *506*
pixels, **497**
in Search Results window, **190–191**
in Source pane, 247
text traits, **500–501**, *501*, 503–504, *504*
editor, **111**
adding text with, **113–114**, 144–145
appearance of, **135–136**, *135*
background of, **111–112**
closing files in, **142**
deleting text in, **120**
Editor window in, **112–113**, *113*
exercises for, **144–157**
fonts in, **122–123**, 148
indenting in, **123–124**, 151–152, *151*
jumping to routines in, **131–132**, *131*
line endings in, **134**, *134*
markers in, **132–134**, *133*, 154, *154*
moving insertion point in, **114–116**
moving text in, **118–119**, *118–119*, 147–148, *147*
multiple windows in, **137–138**
printing files in, **143–144**
punctuation in, **129–131**, 150–151, *151*
related files in, **136–137**, *136*
saving files in, **138–142**, *139*, 145–146
search and replace feature in. *See* search and replace feature
selecting text in, **117–118**, 145
splitting panes in, **124–125**, *125*, 152–153, *152*
syntax coloring in, **126–129**, *127–129*, 148
undoing changes in, **120–122**, 146
Editor Settings panel
for automatic saves, 140
for balancing punctuation, 130
for content menus, 261, *261*
for drag-and-drop, 119, *119*, 149
for multiple undo/redo, 122
for routine sorting order, 132
Editor window, 109, *110*, **112–113**, *113*, 298, *298*
empty oval tool, 495
empty rectangle tool, 494
empty rounded rectangle tool, 494
Enable Debugger command, 332–333, *333*, 336, 390
encapsulation, 217–218, 220
End key in editor, 115
Enter 'Find' String command, 166
Enter key in editor, 114
Enter 'Replace" String command, 167, 173
Entire Word option
for regular expressions, 193
for search and replace feature, 170–171, 202, *203*
Entry Point Usage pop-up menu, 520, 525
Entry Points section, 569–570
enumerated constants, 218, 226–227
Enumeration format command, 373
EOLN Format command, 134
eraser tool, 494–495, 498
errors
breaking on, **366–367**
compiler, **307–314**, *309–311*
handling, **312–313**
linker, 304
messages for, 12, *13*, **310–312**, *311*, 315–319, *316–318*
Errors & Warnings window, 12, *13*, **310–312**, *311*, 315–319, *316–318*

events
frameworks for, 441–442
Log window for, **346**, *346*, 391
exceptions, breaking on, **366–367**
Exer03 folder, 89
Exer04 folder, 144
Exer05 folder, 199
Exer06 folder, 262
Exer07 folder, 314
Exer08 folder, 390
Exer09 folder, 428
Exer10 folder, 457
Exer11 folder, 503
Exer12 folder, 528
Exer13 folder, 543, 549
Exer14 folder, 571
exercises
for browser, **262–274**
for compiling, **313–323**
for Constructor, **502–508**
for debugger, **389–405**
for editor, **144–157**
for frameworks, **456–460**
for menus and toolbar, **36–40**
for projects, **87–101**
for search and replace feature, **199–212**
for standard libraries, **428–436**
exporting functions for DLLs, **526–527**, *527*
Expression window, **375–376**, *375*, 403–404, *404*
extensions for compilers, 78, 283–286

F

Far options, 562
Fast Link option, 564
fat applications, 556, 588
file changed icon
in Editor window, *113*
in Errors & Warnings window, 312
File menu commands, 33, 49, *50*
Close, 60, 142
Close All, 142
New, 109
New Project, 50, 88, 96
New Project File, 469
Open, 58, 68
Open File, 110
Open Selection, 137
Print, 144
Print Setup, 143
Revert, 141
Save, 139, 145
Save a Copy As, 60, 141, 191, 313
Save All, 139
Save As, 140–141
Save Settings, 378
File name extension column, 282
File Name option
for Mac OS programming, 559
for Windows programming, 515
file names in Project window, 55, *55*
file sets, **182**
choosing, **184**
creating, **182–183**, *183*
deleting, **184–185**, *184*
saving, 182, 206–207, *207*
File Sets menu commands, 176
Remove a File Set, 184
Save this File Set, 182, 206
File type column, 282
file type for applications, **576**
files
adding to projects, **62–64**, *64*, 93–95, *93–95*
binary, 100, 288
closing, **142**
for compiling
searches for, 295–297, *295*
selecting, **286–287**, *287*
creating, **109**, *110*
deleting, **66–67**
header. *See* header files
listing, 68, *68*

moving, **66**, *66*
opening, **67–68**, *68*, 99, **110–111**, 136–137, *137*, 149–150, 247, *248*
prefix, 292–293, 301, 321–323, *322*
printing, **143–144**
related, **136–137**, *136*
renaming, **140–141**
reverting, **141**
saving, **138–142**, *139*, 145–146
searching in, **175–185**, *177–179*, *181*, *183–184*
selecting
for compiling, **286–287**, *287*
for debugger, 333, *334*
for projects, **61–62**, *62*
touching, **70–71**, *71*
tracking. *See* projects and project manager
windows for, **108–109**
filled oval tool, 495
filled rectangle tool, 494
filled rounded rectangle tool, 494
filters in single-class browser, 250–252, 270, *271*
final methods, 542
Find command, 162, 171, 186–187, 201, 357
Find all Implementations of command, 255
Find dialog box, 357–358, *357*
Find in Next File command, 182
Find Next command, 165, 171, 200
Find Previous command, 165, 171, 201
Find Previous Selection command, 165
Find Selection command, 164–165, 200
Find window, 162–164, *163*
for batch reports, 186, *186*, 207–209, *208*
for file sets, 206–207, *207*
for finding and replacing text, **171–174**
for multi-file searches, 176–177, *177*, 205–206, *206*
opening, 201, *201*
for range of searches, **169**
for regular expressions, 192–193, *192*, 211–212, *211*
for replacing found items, 204–205, *205*
for scope of searches, **170–171**
for search and replace strings, **166–168**, *167*, 202, *202*
for whole words, 202, *203*
for wrapping in searches, 203–204
finding
code, **357–358**, *357*
editor text. *See* search and replace feature
files for compilation, 295–297, *295*
Fixed format command, 373
Flashing Delay field, 130
floating-point digits for variable values, 372
Floating Point format command, 373
floating-point math routines
C library for, 416
Pascal library for, 424
floating-point unit (FPU) registers, 382
flow control, frameworks for, 442
focus
in class browser, 241
in Search Results window, 191
focus bar for moving files, 66
folders
braces for, 82
in CodeWarrior Gold, 600–605
for installation, 29
parentheses for, 83, 87, 297
for projects, 53
for shortcuts, 30
Font & Tabs panel, 122–123, *122*, 148, *148*
FontEdit.exe utility, 601
fonts
in editor, **122–123**, 148
for menu items, 492
saving settings for, 141
for text, 500, 503–504, *504*
utility for, 601
foreground color for pixels, 495–496
Fract format command, 373
frameworks, **17**, **440**
exercises for, **456–460**
fundamentals of, **17–19**, **441–442**
Java, **447–451**, *448–449*, *451*
MFC, **443–446**, *443*, *446*
PowerPlant, **451–456**, *454*
front-end compilers
settings for, **290–293**, *291*, *293*
tasks for, 21

Full Path in SYM Files option, 564
function pop-up menu, 312
function prototypes
in class browser, 248–249, *250*
as declarations, 227
functions. *See* routines
Functions.p file, 428
Functions pane
in multi-class browser, **244–246**, *246*
in single-class browser, **250–251**, *251*

G

Gdi32.lib file, 94
general utilities
C library for, 415
C++ library for, 420
Generate A6 Stack Frames option, 564
Generate CodeView Info option, 334, 521, 525, 529
Generate Link Map option
for Mac OS programming, 564
for Windows programming, 521
Generate Make Map File option, 80
Generate SYM File option
for Mac OS programming, 564
for Windows programming, 306, 334, 521
Global Optimization option, 568
global variables
in browser, 218, 226–227
in debugger, 357, **367–368**, *368*, 397–398, *398*
windows for, 374–375
Globals pane, 367–368, *368*, 397
Go Back command, 258, 273
Go Forward command, 258, 273
Go-To-Line button
in Editor window, *113*
in Errors & Warnings window, 312
Go To Line Number dialog box, 115, *115*
graphical browser
deep and wide disclosures in, **233–234**, *234*
for multi-class hierarchies, **228–238**, *229–230*, *234–237*
multiple inheritance in, **234–235**, *235–236*
showing and hiding classes in, **230–232**, *231–232*
for single-class hierarchies, 236, *237*, **238–239**, *239*, 266–268, *267*
graphical user interfaces (GUIs)
C and Pascal support for, 414
Constructor as builder of, **464–465**
frameworks for, **441–442**
Java classes for, 447
grep, 192
grids for layout panes, **481–483**, *482–483*
Group names in Project window, 55, *55*
groups
in Constructor, 467
creating and naming, **64–65**, *65*, 95, *96*
deleting, **66–67**
listing files in, 68, *68*
moving, **66**, *66*
searching in, **175–185**, *177–179*, *181*, *183–184*

H

.h extension, 286
hard drives and compilation speed, 297
header files, 5
including, **69–70**, *69*, 99–100, *99*, 418, 430–431, *431*
in MFC, **444–445**
in PowerPlant, **454**, *454*
precompiled, 56, 284, **300–303**, *301–302*, 319–321, *321*
prefix files for, 292–293, 301, 321–323, *322*
searching in, **177–179**, 206
Headers directory
for Mac OS, 604
for Windows, 605
heap
in Mac OS programming, 559
monitoring, 601
in Windows programming, 516
Hello.c file, 573

Hello.cwp file, 571
Hello.rsrc file, 573
(Help) directory, 602
help for toolbar icons, 34, 38
Help menu, 33
hexadecimal digits for variable values, 371
Hexadecimal format command, 373
Hide Toolbar command, 38
hiding
  classes, **230–232**, *231–232*
  data, **219–220**
  errors and warnings, 311
  grids, 481–482
  icons, 135
  toolbar, 35, 38
hierarchical variables, **368–369**
hierarchies
  of classes. *See* multi-class hierarchy view; single-class hierarchy view
  in layouts, **477–479**, *478–479*
highlighting, 4
hits in searches, 164, 189–191
home.htm file, **600**
Home key in editor, 115
horizontal orientation in class browser, 242
hosts, 5
hot spot tool, 495
.html extension, 544
human interface model, **116**
Hw32.exe utility, 601
hyphens (-)
  for breakpoints, 359, 398
  in regular expressions, **196**
  in Source Browser window, 342–343

I

icons in editor, **135**, *135*
Ide.exe file, 601
IDs
  for menu items, 487, 490, *491*, 507
  for menus, 489–490
  for resources, 472–473
  for submenus, 492
Ignore Case option, 170–171
Ignored By Make flag, 283–285
implementations, 5
Import "Weak" command, 449–450, 546
importing Java classes, **450–451**, *451*
In Variable Panes, Show Variable Types By Default option, 372
#include statements and including
  for C libraries, 418, 430–431, *431*
  compilation options for, 297
  header files, **69–70**, *69*, 99–100, *99*
  for MFC, **444–445**
  for PowerPlant, 454
  precompiled files, 56, 284, **300–303**, *301–302*, 319–321, *321*
  syntax choices for, 83
indenting in editor, **122–124**, 151–152, *151*
infinite loops, 351
Info dialog box, 474, *474*
inheritance
  browser display for, **234–235**, *235–236*
  in object-oriented code, 218
  in PowerPlant, **452–455**, *454*
inherited functions, viewing, 251–252, 270, *271*
initialization code
  in Mac OS 68K programming, 559
  in Mac OS PowerPC programming, 569
Initialize Before command, 449–450, 547
inline assembly support, 292, 523
input/output routines
  C library for, 415
  C++ library for, 421
  Java classes for, 447
  Pascal library for, 424
insertion point, 113
  fundamentals of, **114**
  human interface model for, **116**
  moving, **114–116**
  saving position of, 142
installation, **28–31**, *29–31*
installer, 28, *29*
instances, 217–218
Instruction Scheduling option, 568

instruction sets for virtual machines, 23, **538–539**
Integrated Development Environments (IDEs), 6, **14–15**
  for debugger setup, **332–336**, *333–335*
  help files for, 602
interdependencies, project manager for, 7
interface builders, Constructor, **19–20**, *20*
interface files, 5. *See also* header files
interface model, human, **116**
Interface pop-up menu
  in Editor window, *113*, 136, *136*, 287, *287*
  in Errors & Warnings window, 312
  in Search Results window, 190
Interfaces directory, 605
intermediate representation (IR)
  by compilers, **21–22**, 539
  optimizing, **518–520**, *519*
interpreters, Java, 539
interrelated files in editor, **136–137**, *136*
intrinsics support, 292
IR Optimizer panel, 289, **518–520**, *519*
is32BitCompatible flag, 572
italics for help text, 38
iterators, C++ library for, 420

J

Java, **538–539**
  for applets, **543–547**, *544–546*, *548*
  for applications, **548–552**, *549–550*
  classes.zip file for, **448–450**
  debugger with, **331–332**
  framework for, **447–451**, *448–449*, *451*
  importing classes from, **450–451**, *451*
  setting up for, **540–542**, *540–541*
  stationery in, **448**, *448*
  support folder for, 603
Java API Documentation, 447
java.applet package, 447
java.awt package, 447
Java compiler, bytecodes from, 22, 538–539
.java extension, 286, 542
Java Generic Library, 448
java.io package, 447
java.lang package, 447
Java Language Spec, 447
Java Lib.zip file, 549
java.net package, 447
Java Project panel, **541–542**, *541*
Java Support folder, 603
java.util package, 447
Java Virtual Machine (JVM), **22–23**, 331, 447, **538–539**
jumping to routines, **131–132**, *131*
just-in-time (JIT) compilers, 331, 538–539
justification for text, 500
Jview utility, 549, 551–552

K

Kernel32.lib file, 94
keyboard
  for insertion point, 114–115
  shortcuts with, 4, 487
keywords
  color for, 126–127
  editing list of, **126–129**, *128*
Kill command, 348, 351
killing programs, **351**

L

language compilers, 21
  choosing, **282–286**, *283*
  settings for, **290–293**, *291*, *293*
language support, C++ library for, 420
large code model, 562
lasso tool, 494–495
Launchable flag, 283–284
launching
  CodeWarrior, 36
  Constructor, 502–503
  debugger, **336–337**, *338*, 391, *392*

Layout menu commands
Edit Grid, 483
Show/Hide Grid, 481–482
Snap/Don't Snap to Grid, 482
layouts, **476–477**
adding objects to, 505, *505*
containment in, **477–479**, *478–479*
custom panes for, **479–480**, *479*, 504–505
moving panes in, **481–486**, *482–484*
pane properties in, **480–481**, *480*
LBroadcaster class, 452–453
Level option, 568
Lewis, Peter, 543
libraries, **412**
adding, 429–430, *429*
for C, **414–419**, *417*
for C++, **419–423**, *421*
debugging, 350, **384**, 432–435, *433–435*
editing, **426–427**
exercises for, **428–436**
folder for, 604
fundamentals of, **16**, **56–57**, **412–414**
including source code from, 56, 456
for Pascal, **423–426**, *425*
target-dependent, 282
Libraries directory
for Mac OS, 604
for Windows, 605
licenses, 29, 595
Life.bat file, 550
Life game, **543–547**, *544–546*, *548*
Life.html file, 545
Life.java file, 545, 550
LifeApp.cwp file, 549
LifeApp.zip file, 552
LifeApplet.cwp file, 543
Lifetime Analysis option, 520
line endings in editor, **134**, *134*
line numbers for insertion point, 115
line tool, 494, 496
lines (graphics) in browser window, 229, *230*
lines (text)
in regular expressions, **197–198**
selecting, 117
link maps
for Mac OS programming, 564
for Windows programming, 305–306, 521
Link Single Segment option, 564
linked lists, frameworks for, 18
linkers and linking
choosing, **303–304**
with compilers, 281
fundamentals of, 47
for Mac OS programming, **563–565**, *563*, **569–570**, *569*
makefiles for, 7
settings for, **305–306**, *306*
for Windows programming, **520–522**, *521*, 525
lists
of files in groups, 68, *68*
frameworks for, 18
of project files, 60
LListener class, 452–453
local variables
in calling chains, 356
in Program window, 344
in register coloring, **370–371**
viewing values of, **367–369**, *368*
windows for, 374–375
localization, C++ library for, 420
Log window, **346**, *346*, 391
loop invariants, 363
Loop Invariants option, 520
loops, infinite, 351
low-level debugger features, **380–383**, *381–383*
LPicture class, 505
LString class, 453
luminary.h file, 430–431
LumTime.cp file, 525, 527
LView class, 472

M

Mac-hosted debugger, **578–579**
Mac OS, **556–557**, **565–566**
for applications, **570–578**, *571*, *573–574*, *577*
debugging, 563–564, **578–579**

future of, **557**
moving applications to, **575–576**
PowerPlant for, **451–456**, *454*
PPC Codegen panel for, **567–568**, *567*
PPC Linker panel for, **569–570**, *569*
PPC Project panel for, **566**, *566*
68K CodeGen panel for, **561–563**, *561*
68K Linker panel for, **563–565**, *563*
68K Project panel for, **558–560**, *559*
support directory for, 604
Mac OS Support directory, 604
MacHeaders directory, 604
machine code
in disassembling, **297–299**, *298*
for Windows programming, 518
Machine Code Listing option, 518
macros
for compilers, 299
displaying, 219, 226–227
MacsBug Symbols option, 564
Macutils.hqx file, 578
Main option, 570
Make command, 288, 304–305, 319, 336, 390, 431
Make Transparent command, 499
makefiles, 7, 49
Marker menu commands, *113*, 132, *133*
Add marker, 132, 154, *154*
Remove marker, 133
markers
breakpoints as, 365
in editor, **132–134**, *133*, 154, *154*
marquee tool, 494–495
math.h file, 416
math routines
C library for, 416
Pascal library for, 424
MathLib68K(2i).lib file, 571
MBAR resource, 487
MCWHello file, 574–576
MDEF IDs, 489, 560
members
in class browser, **246**, *247*
in object-oriented code, 217
memory
debugger for, **380–381**, *381*
Pascal library routines for, 424
watchpoints for, **385–386**, *386*
memory attributes for resources, 474
Memory window, 380–381, *381*
Menu Info command, 489
Menu Info dialog box, 489, *489*
Menu Item Info command, 491
menus, 4, **32–33**, *32*, **486–488**, *487*, *491*, 506–508, *507*
for debugger, 339
editing, **490–492**, *491*
exercises for, **36–37**, *37*
making, **488**, 507
menu bars for, **488–490**, *488–489*
submenus, **492**
Merge Compiler Glue Into Segment 1 option, 564–565
Merge Into Output command, 449–450, 547
message windows, background of, **187–188**
methods, 5, 217, 542
Metrowerks C runtime library, 418
Metrowerks Standard Library folder, 604
Metrowerks Web site, 594
MFC 4.2.cwp file, 445
MFC Application stationery, 528
Mfc.lib file, 443–445, 455
(MFC4.2) directory, 605
MFCHeaders file, 444–445
MFCHeaders.pch file, 444–445
Microsoft Foundation Classes (MFC), 17, 440
exercise for, **456–460**
including headers from, **444–445**
precompiling, 300–301
source code in, **445–446**
stationery in, **443–444**, *443*
Minimum Heap Size option, 559
missing breakpoints, **362–363**
modification dates and times, tracking, **70–71**, *71*, 80
modules, 217, 220
Mohr, Brad, 570
monospace fonts, 123
Motorola series processors, 556
mouse with insertion point, 114–115

movable toolbars, 34–35, 37
moving
applications to MAC OS computers, **575–576**
in Constructor, 467
Constructor projects, **475**
files and groups, **66**, *66*
insertion point, **114–116**
layout panes, **481–486**, *482–484*
menu items, 490
menus, 489
text, **118–119**, *118–119*, 147–148, *147*
MPW utility, 560
MSL C.68k(4i).lib file, 417
MSL C library, calling from Pascal, **427–428**
MSL C PPC.lib file, 417
MSL C++.68K.lib file, 422
MSL C++ PPC.lib file, 422
MSL C Reference, 415
MSL C++ Reference, 420
multi-class browsing, **10–11**, *11*, 219, 222–224, **239–242**, *240*, *242*, 268–270, *268–270*
for class declarations, 256
Classes pane in, **242–244**, *243–244*
Data Members pane in, **246**, *247*
for functions, 257
Functions pane in, **244–246**, *246*
Source pane in, **241–242**, *242*, **247–248**, *248*
for viewing and editing code, **248–249**
multi-class hierarchy view, 228–229, *229*, 265–266, *266*
appearance of, **229**, *230*
for class declarations, 256
deep and wide disclosures in, **233–234**, *234*
for functions, 257
for multiple inheritance, **234–235**, *235–236*
navigating in, **235–238**, *237*
showing and hiding classes in, **230–232**, *231–232*
multiple document interface (MDI) windows, 32
multiple files, searching in, **175–185**, *177–179*, *181*, *183–184*, 205–210, *206–210*
multiple inheritance
browser display for, **234–235**, *235–236*
in PowerPlant, **452–455**, *454*
multiple languages, **20**
compilers for, **21–24**, *21*
debugger for, **387**
multiple projects, debugger for, **388–389**
multiple targets, **587–589**, *588*
multiple undo/redo feature, **120–122**
multiple windows, **137–138**
multiply-defined symbols, **252–254**, *253*
multitasking, debugger for, **387–388**, *388*
Mwcrtl.lib file, 94, 418, 429, 444
MWDebug.exe file, 601, 604
MyMFCapp.cp file, 458
MyMFCapp.h file, 458

N

name collisions in Pascal, 426
Name new project as dialog box, 53–54, *54*, 89
Name option, 541
names
for files, **140–141**
for groups, **64–65**, *65*, 95, *96*
for Java programs, 541
for Mac OS programs, 541
for markers, 132
for menus, 491
for projects, 53
for resources, 472–473
for source files, 282
spaces in, 285, 551
for text traits, 501
on title bars, 108
for variables, 372
for Windows projects, 515
narrow-scope browser view, 222–223
Native option, 520
navigating code
in browser, **254–257**
in debugger, **355–358**, *356–357*
network support, Java classes for, 447

New command, 109
New Class Browser command, 223, 240, 268
New Menu command, 488, 507
New Menu Item command, 490, 507
New Project command, 50, 88, 96
New Project dialog box, 50–53, *50*, *53*, 88, 96–97, *97*
New Project File command, 469
New Resource command, 469, *470*, 503
New Separator Item command, 490
NeXT, purchase of, 557
null-terminated strings, 428
number signs (#) for preprocessor statements, 299
numerics, C++ library for, 421

O

object code
    deleting, 100, 288
    generating, 47
object-oriented code
    with browser, 219
    data hiding in, **219–220**
    debugging, 341
    frameworks for, 18, 442
    fundamentals of, **217–218**
Objective C language, 557
objects
    in object-oriented code, 217
    overlapping, **477–479**, *478–479*
    properties of, 505–506, *506*
Open command, 58, 68
Open Array Window command, 377
Open dialog box
    with debugger, 337, *338*
    for projects, 58, *59*
Open File command, 110
Open Selection command, 137
Open Variable Window command, 374
opening
    browser files, 247, *248*
    Find window, 201, *201*
    groups, 65
    project files, **67–68**, *68*, 99, **110–111**, 149–150
    projects, **58–59**, *59*, 100–101
    related files, **136–137**, *136*
operating systems, 5, **280–282**, *281*
Optimize Code Generation option, 541
Optimize For option, 568
Optimize Space option, 518
Optimize Speed option, 518
optimizing
    intermediate representation, **518–520**, *519*
    in Java programming, 541–542
    in Mac OS 68K programming, 562
    in Mac OS PowerPC programming, 568
    missing breakpoints from, **362–363**
    in Windows programming, 518
Options menu commands, *113*
    for line endings, 134
    Make Transparent, 499
    Syntax Coloring, 129, *129*
ordinal operations, Pascal library for, 424
orientation
    in class browser, 242
    for printing, 143
Output File Type option, 515
Output Format option, 541
Output settings for Windows programming, 521–522
overlapping layout objects, **477–479**, *478–479*

P

.p extension, 286
pad-bytes, 517
Page Down key, 115
Page Up key, 115
paint tool, 494, 498–499
Pane Splitter, *113*
panes
    in editor, **124–125**, *125*, 152–153, *152*
    in layouts, **479–480**, *479*
        aligning, **483–485**, *483–484*

custom, 504–505
distributing, **485–486**
moving, **481–486**, *482–484*
properties for, **480–481**, *480*
in multi-class browser
Classes, **242–244**, *243–244*
Data Members, **246**, *247*
Functions, **244–246**, *246*
Source, **241–242**, *242*, **247–248**, *248*
in Search Results window, 190
in single-class browser, **250–252**, *251*
paper size and source, 143
parallel processing, 585
parameters
in class browser, 249, *250*
for DLLs, 57
parentheses ()
balancing, **129–131**
for folders, 83, 87, 297
in regular expressions, **195–196**
parser
in browser, 216, 218
symbol catalog by, 218–219
.pas extension, 286
Pascal
calling MSL C library from, **427–428**
dependency lists for, 80
libraries for, 16, **423–426**, *425*
link maps for, 306
Pascal Compiler panel, **292–293**, *293*
Pascal Library Reference, 423, 425
Pascal String format command, 373
Pascal strings for variable values, 372
Pascal Warnings panel, 309–310, *310*
paths
for applications, **551–552**
in Errors & Warnings window, 312
for projects, **81–84**, *82*, 178, *178*, **293–296**, *294–295*
in Source pane, 247
in symbolics file, 341
pattern tool, 495, 498
patterns in bitmaps, **497–498**
PC Exchange utility, 475, 575
.pch extension, 284
compilers for, 286
including files with, 301–303
.txt with, 320
.pch++ extension, 286
Peephole Optimization option
for Mac OS programming, 568
for Windows programming, 518
pencil tool, 494–495
periods (.) in regular expressions, **194**
pixels
color of, **495–496**
editing, **497**
placeholder files, 85
platforms
converting code between, 134
with debugger, **330–332**
libraries for, 412–413
with PowerPlant, 455
Please Select an Access Path dialog box, 295–296, *296*
Plugin API Guide, 280
plugin compilers, **279–280**, *279*
Plugins folder, 279–280, *279*, 601
plus signs (+) in regular expressions, **195**
Pop-up Display Button, *113*
pop-up menus in Project window, 55, 58
Post Linker pop-up menu, 304
post-linkers, 78, 304
PowerPC programming, 556, **565–566**
PPC Codegen panel for, **567–568**, *567*
PPC Linker panel for, **569–570**, *569*
PPC Project panel for, **566**, *566*
PowerPlant Advanced Topics, 456, 466
PowerPlant Book, 456, 466, 487
PowerPlant directory, 604
PowerPlant framework, 17, 440, **451**, **454–455**, *455*
background of, **452–453**
precompiling, 300–301
PPC Codegen panel, **567–568**, *567*
PPC Linker panel, **569–570**, *569*
PPC Project panel, **566**, *566*
PPob resource, 476, 504–505, *504*

#pragma statements, 134, 299, 523
Precompile command, 301, 321
precompiled files, 56, 284
creating, **300–303**, *301–302*, 319–321, *321*
in MFC, **444–445**
prefix files for, 292–293, 301, 321–323, *322*
Precompiled flag, 282, 284
precompiling resources, 522
Preference dialog box, 38, *39*, 74–75, *74*
for automatic saves, 140
for balancing punctuation, 130
for browser, 260–261, *260–261*
for debugging, 349–351, *349*
for drag-and-drop, 119, *119*, 149
for fonts, 122, *122*, 148, *148*
for multiple undo/redo, 122
for routine sorting order, 132
for syntax coloring, 126–127, *127*
for unbounded arrays, 377, *378*
Preferences command, 38, 74, 119
preferences for projects, **73–75**, *74*
Preferred Heap Size option, 559
prefix files
for precompiled headers, 301, 321–323, *322*
support for, 292–293
Preprocess command, 299
preprocessors for compilers, **299–300**, *300*
Print command, 144
Print Setup command, 143
Print Setup dialog box, 143–144, *143*
printing files, **143–144**
private members option, 251
private methods, 542
procedures, 5
processes, 5
Processes window, 388, *388*
processors
Mac OS programming options for, 562
target compilers for, **280–282**, *281*
Produce.cwp file, 262
profiler calls, setting for, 563
Profiler utility, 578, 604–605
Program Control panel, 349–351, *349*
program counter, current statement arrow as, **353–355**, *354*, *394*
Program window, 337, *338*, **343–345**, *343*, 368, *368*, 391, *392*
progress indicator for compilation, 294, *294*
Project Headers option, 177–178, 206
Project menu commands, 33
Add Files, 64, 94, 321, 417, 429, 457–458
Add Window, 63, 93
Bring Up To Date, 98, 155, 262–263, 288, 315
Check Syntax, 289
Compile, 288, 314
Debug, 288, 304–305, 336–337, 432
Disassemble, 298
Enable Debugger, 332–333, *333*, 336, 390
Make, 288, 304–305, 319, 336, 390, 431
Precompile, 301, 321
Preprocess, 299
Remove Binaries, 100, 288
Remove Files, 67, 458
Run, 288, 304–305, 319, 336
Synchronize Modification Dates, 71
project relative access paths, 296
Project Settings command, 76, 89, 127
Project Settings dialog box
for access paths, 81–84, *82*, 178, *178*, 293–296, *295*
for Build Extras, 79–80, *81*, 224–225, *225*
for C/C++ compilers, 290–292, *291*
for compiler, 11, *12*
for debugger, 334–336, *335*
for Java programming, **540–542**, *540–541*
for keywords, 128–129, *128*
for linkers, 305–306, *306*
for Mac OS 68K programming, **558–565**, *559*, *561*, *563*
for Mac OS PowerPC programming, **565–570**, *566–567*, *569*
for Pascal compilers, 292–293, *293*
for targets, 77–79, *77*, *80*, 89–92, *92*, 280–286, *281*, *283*
for warnings, 308–310, *309–310*
for Windows programming, **515–522**, *516–517*, *519*, *521*
Project Switch List command, 59, *59*, 100–101, *101*
project totals, 55, *55*, 58
Project Type option, 558–559
Project window, **54–58**, *55*

projects and project manager, **46–48**, *47*
access paths for, **81–84**, *82*, 178, *178*, 293–294, *294*
adding files to, **62–64**, *64*, 93–95, *93–95*
adding libraries to, 429–430, *429*
background of, **49**
binary files in, 100
Build Extras settings for, **79–80**, *81*
C library in, **416–419**, *417*
C++ library in, **421–423**, *421*
closing, **60–61**, 100
copying, 60
creating, **49–54**, *50*, *53–54*, 88–89, 96–97, *97–98*
debugging information for, **71–73**, *72*
default options for, **75–76**
elements in, 6, *6*
exercises for, **87–101**
groups for, **64–67**, *65–66*, 95, *96*
included files in, **69–70**, *69*, 99–100, *99*
new version of, **587–589**, *587–588*
opening, **58–59**, *59*, 100–101
opening files in, **67–68**, *68*, 99, **110–111**, 149–150
Pascal library in, **425–426**, *425*
preferences for, **73–75**, *74*
Project window for, **54–58**, *55*
saving, 54, **60**, 89
selecting files in, **61–62**, *62*
settings for, **76**, **79**, *80*
stationery for, **84–87**, *86*, 96, *97*
targets for, **77–79**, *77*, *80*, **89–92**, *92*, 280–286, *281*, *283*, **587–589**, *588*
touching files in, **70–71**, *71*, 287
updating, **98–99**, *98*, 155, 315
properties
for menu items, 490–491, *491*
for objects, 505–506, *506*
for panes, **480–481**, *480*
property inspector window
for menu items, 491, *491*
for panes, 480, *480*
protected option, 251
prototypes
in class browser, 248–249, *250*
as declarations, 227
public option, 251
punctuation, balancing, **129–131**, 150–151, *151*
pure virtual functions, 245

Q

quotation marks (") for included files, 83, 297

R

\r character
in regular expressions, 198
in searches, 168
ranges
of memory, watchpoints for, **385–386**, *386*
in regular expressions, **196**
with search and replace, 162, **169**, **180–181**, *181*
of text, selecting, **117–118**
rapid application development (RAD) tool. *See* Constructor
.rc files, 284
recursive file searches, 295–297, *295*
red check marks in editor, 139, *139*, 144, *145*
red circles for breakpoints, 359, 361
Redo command, 120–121, 146
Reduction in Strength option, 520
Regexp option, 168, 192–193
register coloring, **370–371**
registers
A6, 564
viewing and changing, **382**, *382*
Registers window, 382, *382*
registration, 595–596
regular expressions, **191–192**, 210–212, *211*
activating, **192–193**, *192*
ampersands in, **198–199**
asterisks in, **194–195**
backslashes in, **197–198**
brackets and hyphens in, **196**

carets in, **196–197**
dollar signs in, **197–198**
parentheses in, **195–196**
periods in, **194**
plus signs in, **195**
syntax of, **193–194**
working with, **199**
related files in editor, **136–137**, *136*
relative access paths, 296
release notes, folder for, 603
Remove a File Set command, 184
Remove Binaries command, 100, 288
Remove Files command, 67, 458
Remove marker command, 133
Remove Markers dialog box, 133, *133*
removing. *See* deleting
renaming
files, **140–141**
groups, 65
repeating code, **353–355**, *354*
repetitions in regular expressions, **194–196**
Replace command, 173, 190
Replace All command, 173–174, 181
Replace & Find Next command, 173
Replace & Find Previous command, 173
replacement text in search and replace, 161–162, **166–168**, *167*
replacing elements. *See* search and replace feature
.res extension, 284, 522
ResEdit editor, 576
Reserved Heap Size option, 516
Reserved Stack Size option, 516
Reset Toolbar command, 39
resetting toolbar, 35, 39
resize bars, 312
resource compiler, **522**
Resource File flag, 283–284
Resource.frk folder, 91, **474–476**, 575–576
Resource Info command, 474
resources
in Constructor, 466–467
directory for, 604
making, **469–472**, *470–471*
precompiling, 522
and resource forks, 91, **474–476**
working with, **472–474**, *474*
Resources directory, 604
return characters
in regular expressions, 198
in searches, 168
reusing code. *See* libraries
Revert command, 141
reverting files, **141**
right-angle lines in browser window, 229, *230*
RISC (reduced instruction set chips), 565
Routine pop-up menu
in Editor window, *113*, 131, *131*, 153, *153*
in Search Results window, 190
in Source Browser window, 342
routines, 4–5
in browser, 218, 226–227, **244–246**, *246*, **256–257**
calling chains for, **344**
defined, 5
jumping to, **131–132**, *131*
libraries for. *See* libraries
stepping into, **352–353**, 394–395, *396*
stepping out of, **353**
stepping over, **352**, 393
Routines.c file, 149, *149*, 152, *152*, 155–156, 199–201
Routines.h file, 155–156
.rsr extension, 469, 475
.rsrc extension, 573
Run command
for debugger, 343, 348, 350
for projects, 288, 304–305, 319, 336
running
applets, **547**, *548*
applications, **530–531**, **552**, **577**, *577*
debugger code, **348–350**, *349*

S

sales department, **595**
sample bitmap views, **496–497**

Save command, 139, 145
Save a Copy As command
for Errors & Warnings window, 313
for hit lists, 191
for projects files, 60, 141
Save All command, 139
Save All Before "Update" option, 140
Save As command, 140–141
Save Default Window command, 136, 142, 150, 269
Save File Set As dialog box, 182, *183*
Save Settings command, 378
Save Settings in Local ".dbg" Files option, 379
Save this File Set command, 182, 206
saving
assembly-language code, 298
automatic, **139–140**
changes, 109
debug information, **378–379**
Errors & Warnings window, 313
file sets, 182, 206–207, *207*
files in editor, **138–142**, *139*, 145–146
projects, 54, **60**, 89
window settings, **141–142**
scaling bitmaps, 497
scope with search and replace, 162, **170–171**
scroll bars in panes, 125
search and replace feature, **160–161**
adding files for, **179**
batch reports for, **185–191**, *186*, *189*, 207–210, *208–210*
controls for, **162–164**, *163*
deleting files from searches, **180**
design of, **161–162**
exercises for, **199–212**
file sets for, **182–185**, *183–184*, 206–207, *207*
without Find window, **174–175**
finding and replacing text in, **171–174**, 200–203, *200*, *202–203*
for multiple files, **175–185**, 205–210, *206–210*
ranges with, 162, **169**, **180–181**, *181*
regular expressions for, **191–199**, *192*, 210–212, *211*
scope with, 162, **170–171**
for selected text, **164–166**
selecting files for, **176–178**, 205–206, *206–207*
for single files, **166–175**, *167*
special characters in, **168**
Search menu commands, 33, 162, *163*
Enter 'Find' String, 166
Enter 'Replace" String, 167, 173
Find, 162, 171, 186–187, 201
Find in Next File, 182
Find Next, 165, 171, 200
Find Previous, 165, 171, 201
Find Previous Selection, 165
Find Selection, 164–165, 200
Go Back, 258, 273
Go Forward, 258, 273
Replace, 173, 190
Replace All, 173–174, 181
Replace & Find Next, 173
Replace & Find Previous, 173
Search Results window, **187–190**, *189*, 208–210, *209–210*
background of, **187–188**
editing in, **190–191**
search strings, **166–168**, *167*
searches
in code, **357–358**, *357*
in compilation, 295–297, *295*
in editor. *See* search and replace feature
Select files to add dialog box, 64, *64*, 94, 429, *429*
Select files to search dialog box, 179, *179*
Select Type dialog box, 381, *381*
selected text, finding, **164–166**
selecting
vs. choosing, 4
in Constructor, 467
files in projects, **61–62**, *62*
text, **117–118**, 145
semicolons (;)
compiler warnings for, 309
errors involving, 316–317
Sentence.cwp file, 390–392, *392*
separator bars for menus, 490
Set Watchpoint command, 385
settings, 73

Settings & Display panel, 341
for data types, 372
for saving debug information, 379
for unbounded arrays, 377, *378*
Settings dialog box. *See* Project Settings dialog box
shadows for resource forks, **91**, 474
shared libraries, 560
Shift-clicking, 117
Shift key
with drawing tools, 496
in selecting files, 61
in selecting text, 117–118
Shift Left command, 123, 152
Shift Right command, 123, 152
shifting indentation, **123–124**, 151–152, *151*
shortcuts, installation of, 29–30
Show Breakpoints command, 364, 400
Show Catalog Window command, 223, 225
Show Declaration option, 271
Show Expressions command, 375, 403
Show FPU Registers command, 382
Show/Hide Grid command, 481–482
Show Hierarchy option, 271
Show Hierarchy Window command, 223, 228
Show Processes command, 387–388
Show Registers command, 382
Show Toolbar command, 37
Show Types command, 372, 402
Show Watchpoints command, 386
sibling classes, displaying, 229, 233
Signed Decimal format command, 373
single-class browsing, **249–252**, *251*, 270–272, *271–272*
for class declarations, 256
for functions, 257
single-class hierarchy view, 236, *237*, **238–239**, *239*, 266–268, *267*
for class declarations, 256
for functions, 257
single files, searching in, **166–175**, *167*
site licenses, 595
68K CodeGen panel, **561–563**, *561*
68K Linker panel, **563–565**, *563*
68K Project panel, **558–560**, *559*
size
of fonts, **122–123**
of layout grids, 483, *483*
of panes, 125, 241
of paper, 143
of windows, saving settings for, 141–142
'SIZE' flags option, 559, 572
skipping code, **353–355**, *354*
SlideShow.rsr file, 503
small code model, 562
smart code model, 562
Snap/Don't Snap to Grid command, 482
Solution Code folder, 87
Sort Function Popup option, 132
Sort Functions by Method Names in Browser option, 341
sorting
classes in Class pane, 243, *243*
routine names, 131–132
Source Browser window
for code, **356–357**
in debugger, 337, *338*, **340–343**, *340*
for variable values, 367–368, *368*
source files
for applets, **544–545**, *545*
for applications, **529–530**, *530*, **550**, *550*, **572–574**, *573*
defined, 5
for DLLs, **525–526**, *526*
editor for. *See* editor
including, 456
for Java programming, 541
for libraries, 418, **426–427**
in MFC, **445–446**
for multiple targets, 589
names for, 282
searching in, **177–179**, 206
switching with debugger, **358–359**
source-level debuggers, 328
Source Model pop-up menu, 291
Source pane
in class browser, **241–242**, *242*, **247–248**, *248*
in Errors & Warnings window, 311–312, *311*
in Source Browser window, 342

Sources search option, 177, 206
spaces
in directory names, 551
in file names, 285
special characters
in regular expressions, **198**
in searches, **168**
speeding up compiling, **293–297**, *294–296*
splitting
groups, 65
panes, **124–125**, *125*, 152–153, *152*
Spread pane distribution options, 486
square brackets ([])
balancing, **129–131**
in regular expressions, **196**
squares, drawing, 496
Stack command, 137–138
stack crawl pane
examining, 350
fundamentals of, **344**
walking through, **355–356**, *356*, 395, *396*
stack frames, 564
Stack pane, 395, *396*
stack pointer, 564
stacks
and calling chains, 344
size of, 516
standard libraries. *See* libraries
standard object model (SOM), 291
Standard Template Library (STL), 419, 421, 423
Start Code folder, 87
Startup Code option, 559
static constructors, debugging, 349
static data members, 246
static functions, 245
static-link libraries, **56–57**
static methods, 542
stationery, **50–53**, *53*
background of, **52**
customizing, **84–87**, *86*, 96, *97*
directory for, 603
in Java, **448**, *448*
in MFC, **443–444**, *443*
for targets, 51–52, 281
(`Stationery`) folder, 86–87, *86*, 96, *97*, 603
stdio.h file, 415
stdlib.h file, 415
Step Into command, 348, 352–353, 394
Step Out command, 348, 353
Step Over command, 348, 352, 393
Stepanov, Alexander, 419
stepping
through code, **347–355**, *347*, *353–354*
into routines, **352–353**, 394–395, *396*
out of routines, **353**
over routines, **352**, 393
STL Tutorial and Reference Guide, 421
Stop at Beginning of Main When Launching Applications option, 350
Stop at End of File option, 182
Stop command, 348, 350–351
stopping programs, **350–351**
Store Analysis Results option, 80
Stress.exe utility, 601
string.h file, 416
strings
C library for, 416
C++ library for, 420
C-style vs. Pascal-style, **428**
color for, 126
Pascal library for, 424
for variable values, 372
Struct Alignment option
for Mac OS 68K programming, 562
for Mac OS PowerPC programming, 567
structures
aligning, 562, 567
frameworks for, 18
viewing values of, **368–369**
student pricing, 595
styles for text, 500
subclasses in browser window, 228–229
Submenu ID field, 492
submenus, **492**

subroutines. *See* routines
SubSystem IDs, 521
SubSystem pop-up menu, 92, 520–521
support, **594–595**
Suppress Warning Messages option, 564
switching
  between debugger and source code, **358–359**
  panes, 125
  projects, 59, *59*, 100–101
.SYM extension, 329, 336
SYM files, 305–306, 329, 390–391
  for Mac OS programming, 564
  for Windows programming, 518
SYM Format option, 518
Symbol window
  in browser, **252–254**, *253*, 272–273, *273*
  for functions, 257
symbolics file
  for debugger, 328–329
  paths in, 341
symbols
  in browser, 218, 226–227, **252–254**, *253*
  context menu for, 238
Synchronize Modification Dates command, 71
syntax coloring, **126–129**, *127–129*, 148
Syntax Coloring command, 129, *129*
Syntax Coloring panel, 126–127, *127*
system header files, searching in, **177–179**
System Headers option, 177–178
System Include Path option, 81–82
System qualifier, 426
system relative access paths, 296

T

\t character
  in regular expressions, 198
  in searches, 168
tab characters
  in regular expressions, 198
  in searches, 168
  setting number of spaces for, 122
Tab key
  in class browser, 241
  in Constructor, 468
  in Errors & Warnings window, 316, *317*, 318
  in Search Results window, 191
tables, frameworks for, 18
target compilers, **280–282**, *281*
Target panel, 11, *12*, 77–79, *77*, 89–92, 280–286, *281*, *283*, **540**, *540*
target text in search and replace, 161–162
Targeting Java, 447
Targeting Mac OS, 560, 562, 567
Targeting Win32, 292, 523, 527
targets, **20–24**, *21*
  defined, 5
  multiple, **587–589**, *588*
  for projects, **77–79**, *77*, *80*, **89–92**, *92*, 280–286, *281*, *283*
  stationery for, **51–52**
technical support, **594–595**
template classes in C++ library, 419
templates
  displaying with browser, 219, 226–227
  stationery for. *See* stationery
temporary breakpoints, **360–361**, 394, *395*
Termination option, 570
termination routines, Pascal library for, 424
tests in conditional breakpoints, 366
text
  adding, **113–114**, 144–145
  color for, **126–129**, *127–129*, 148, 150, 500
  in Constructor, 468, **500–501**, *501*, 503–504, *504*
  deleting, **120**
  moving, **118–119**, *118–119*, 147–148, *147*
  for search and replace strings, **166–168**, *167*
  searching for. *See* search and replace feature
  selecting, **117–118**, 145
text tool, 494
text traits, **500–501**, *501*, 503–504, *504*
third-party utilities, 601–602
Tile command, 137–138
Tile Vertical command, 137–138, 155–156

tiling windows, **137–138**, 155–156
time functions, C library for, 416
time.h file, 416
Timestep.cp file, 525, 527
title bars, 108
titles of menus, 489
toolbars, 4, **33–35**, *34*
  clearing, 39
  exercises for, **37–40**, *38–39*
  resetting, 35, 39
Tools directory, 604–605
Tools menu commands, 33–34, *35*
  Clear Toolbar, 39
  Hide Toolbar, 38
  Reset Toolbar, 39
  Show Toolbar, 37
  Unanchor Toolbar, 35, 37
Touch column, *55*, 70–71, *71*
Touch command, 69, 71, 287
touching files, **70–71**, *71*
tracking files. *See* projects and project manager
trailing spaces in file names, 285
transparency in bitmaps, **499–500**, *499*
Treat All Warnings as Errors option, 314
Treat #include <...> as #include "..." option, 83, 297
triangles
  disclosure. *See* disclosure triangles
  for multiple inheritance, 234
  in PowerPlant, 454
TrivialApplet.html file, 543
TrivialApplet.java file, 543
TrivialApplication.bat file, 549
TrivialApplication.java file, 549
Turbo Pascal language extensions, 523
.txt extension with .pch, 320
type-ahead selection, 62
Type option
  for Java programming, 541
  for Mac OS programming, 559
typedefs in browser, 219, 226–227

U

unacceptable characters in regular expressions, **196–197**
Unanchor Toolbar command, 35, 37
unbounded arrays, 377, *378*
Undo command, 120–121, 146
undoing changes, **120–122**, 146
Unsigned Decimal format command, 373
untitled windows, 109
Untouch command, 69, 71, 287
Unused Arguments warnings, 308
Unused Variables warnings, 308
updating
  CodeWarrior, 595–596
  DLLs, 57
  projects, **98–99**, *98*, 155, 315
Use Modification Date Caching option, 80
Use Multiple Undo option, 122
User Include Path option, 81–83, 297
User Specified option, 520
User32.lib file, 94
UserID, 521
USES clause, 427
Utility.cp file, 525

V

variables
  array, **376–377**, *376–377*, 402–403, *403*
  in breakpoint conditions, 366, 401, *401*
  in browser, 218, 226–227
  in calling chains, 356
  data types for, **372–374**, *373*, 401–403, *402–403*
  in debugger, 357, **367–368**, *368*, 397–398, *398*
  Expression window for, **375–376**, *375*, 403–404, *404*
  in object-oriented code, 217
  in optimizing, 363
  in Program window, 344, 368, *368*, 394–395, 397
  in register coloring, **370–371**

setting values of, **369–372**
in Source Browser window, 357
watchpoints for, **384–387**, *386–387*
windows for, **374–375**, *374*
Variables pane, 368, *368*, 394–395, 397
vertical orientation in class browser, 242
video capture utility, 601
View Memory command, 380, 385
View Memory As command, 381
View Type pop-up menu, 470, *471*
viewing toolbar, 35
views
for bitmaps, **496–497**
in browser, **222–224**, *223*, **254–255**
in Constructor, 465
virtual functions, 245
virtual machines, **22–23**, **538–539**
Visual C++, CodeView format with, 337
visual interface in browser, 216, 219

W

walking
through calling chains, **355–356**, *356*, 395, *396*
through code, **355**, *356–357*
warnings, 12, *13*, **307–308**, **313–314**
Errors & Warnings window for, **310–312**, *311*, 315–319, *316–318*
in Mac OS programming, 564
settings for, **308–310**, *309–310*
watchpoints, **384–385**
setting and clearing, **385–386**, *386*
viewing, **386–387**, *387*
Watchpoints window, 386–387, *387*
Web sites
code for, 600
Metrowerks, 594
whole words
with regular expressions, 193
in searches, 170–171, 202, 358
wide disclosures in browser, **233–234**, *234*
wide-scope view in browser, 222–224
Win32-x86 Support folder, **605**
Win32ResHeaders++ file, 522
Win32x86 target, 289–290, 305
Window menu commands, 33
New Class Browser, 223, 240, 268
Project Switch List, 59, *59*, 100–101, *101*
Save Default Window, 136, 142, 150, 269
Show Breakpoints, 364, 400
Show Catalog Window, 223, 225, 264
Show Expressions, 375, 403
Show FPU Registers, 382
Show Hierarchy Window, 223, 228
Show Processes, 387–388
Show Registers, 382
Show Watchpoints, 386
Stack, 137–138
Tile, 137–138
Tile Vertical, 137–138, 155–156
Zoom, 137–138
windows
in Constructor, 465
cycling through, 138, 155
for files, **108–109**
moving text between, 148
multiple, **137–138**
saving settings for, **141–142**
splitting, **124–125**, *125*, 152–153, *152*
tiling, **137–138**, 155–156
Windows CUI option, 520
Windows GUI option, 520
Windows menu command, Alignment Palette, 483
Windows programming, **514**
for applications, **528–531**, *529–530*
debugging, **531–534**, *532–533*, *535*
for DLLs, **523–528**, *524*, *526–527*
features in, 522–523
IR Optimizer panel for, **518–520**, *519*
setting up for, **514–515**
WinRC Compiler panel for, **522**
x86 CodeGen panel for, **517–518**, *517*

x86 Linker panel for, **520–522**, *521*, 525
x86 Project panel for, **515–516**, *516*
Windows SDK, precompiling, 300–301
WinRC Compiler panel, 289, **522**
WinRes Import tool, 522
word wrap in editor, 114
words
with regular expressions, 193
in searches, 170–171, 202, *203*, 358
selecting, 117
Wrap option, 169, 181, 203
wrapping
in editor, 114
in searches, 169, 173–174, 181, 203–204, *204*, 358
`www.metrowerks.com` site, 594

X

x86 CodeGen panel, 289, 334, *335*, 390, **517–518**, *517*
x86 Linker panel, 92, *92*, 289, 305–306, *306*, 334, *335*, 390, **520–522**, *521*, 525
x86 Project panel, 79, *80*, 90, 289, **515–516**, *516*

Z

zip files for Java output, 542, **546–547**
ZoneRanger utility, 578, 604–605
Zoom command, 137–138

Metrowerks End-User Software License

PLEASE READ THIS LICENSE CAREFULLY BEFORE USING THE SOFTWARE. BY USING THE SOFTWARE, YOU ARE AGREEING TO BE BOUND BY THE TERMS OF THIS LICENSE. IF YOU DO NOT AGREE TO THE TERMS OF THIS LICENSE, PROMPTLY RETURN THE UNUSED SOFTWARE TO THE PLACE WHERE YOU OBTAINED IT AND YOUR MONEY WILL BE REFUNDED.

1. **License.** The application, demonstration, system and other software accompanying this License, whether on disk, in read only memory, or on any other media (the "Software") the related documentation and fonts are licensed to you by Metrowerks. You own the disk on which the Software and fonts are recorded but Metrowerks and/or Metrowerks' Licensors retain title to the Software, related documentation and fonts. This License allows you to use the Software and fonts on a single computer and make one copy of the Software and fonts in machine-readable form for backup purposes only. You must reproduce on such copy the Metrowerks copyright notice and any other proprietary legends that were on the original copy of the Software and fonts. You may also transfer all your license rights in the Software and fonts, the backup copy of the Software and fonts, the related documentation and a copy of this License to another party, provided the other party reads and agrees to accept the terms and conditions of this License.

2. **Restrictions.** The Software contains copyrighted material, trade secrets and other proprietary material. In order to protect them, and except as permitted by applicable legislation, you may not decompile, reverse engineer, disassemble or otherwise reduce the Software to a human-perceivable form. You may not modify, network, rent, lease, loan, distribute or create derivative works based upon the Software in whole or in part. You may not electronically transmit the Software from one computer to another or over a network.

3. **Termination.** This License is effective until terminated. You may terminate this License at any time by destroying the Software, related documentation and fonts and all copies thereof. This License will terminate immediately without notice from Metrowerks if you fail to comply with any provision of this License. Upon termination you must destroy the Software, related documentation and fonts and all copies thereof.

4. **Export Law Assurances.** You agree and certify that neither the Software nor any other technical data received from Metrowerks, nor the direct product thereof, will be exported outside the United States except as authorized and as permitted by the laws and regulations of the United States. If the Software has been rightfully obtained by you outside of the United States, you agree that you will not re-export the Software nor any other technical data received from Metrowerks, nor the direct product thereof, except as permitted by the laws and regulations of the United

States and the laws and regulations of the jurisdiction in which you obtained the Software.

5. **Government End Users.** If you are acquiring the Software and fonts on behalf of any unit or agency of the United States Government, the following provisions apply. The Government agrees: (i) if the Software and fonts are supplied to the Department of Defense (DoD) , the Software and fonts are classified as "Commercial Computer Software" and the Government is acquiring only "restricted rights" in the Software, its documentation and fonts as that term is defined in Clause 252.227-7013(c) (1) of the DFARS; and (ii) if the Software and fonts are supplied to any unit or agency of the United States Government other than DoD, the Government's rights in the Software, its documentation and fonts will be as defined in Clause 52.227-19(c) (2) of the FAR or, in the case of NASA, in Clause 18-52.227-86(d) of the NASA Supplement to the FAR.

6. **Limited Warranty on Media.** Metrowerks and Sybex warrants the diskettes and/or compact disc on which the Software and fonts are recorded to be free from defects in materials and workmanship under normal use for a period of ninety (90) days from the date of purchase as evidenced by a copy of the receipt. Metrowerks' and Sybex's entire liability and your exclusive remedy will be replacement of the Sybex's diskettes and/or compact disc not meeting Metrowerks' and Sybex limited warranty and which is returned to Metrowerks or Sybex or a Metrowerks authorized representative with a copy of the receipt. Metrowerks and Sybex will have no responsibility to replace a disk/disc damaged by accident, abuse or misapplication. ANY IMPLIED WARRANTIES ON THE DISKETTES AND/OR COMPACT DISK, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF DELIVERY. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS, AND YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY BY JURISDICTION.

7. **Disclaimer of Warranty on Software.** You expressly acknowledge and agree that use of the Software and fonts is at your sole risk. The Software, related documentation and fonts are provided "AS IS" and without warranty of any kind and Metrowerks and Metrowerks' Licensor(s) (for the purposes of provisions 7 and 8, Metrowerks and Metrowerks' Licensor(s) shall be collectively referred to as "Metrowerks") EXPRESSLY DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. METROWERKS DOES NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE SOFTWARE WILL MEET YOUR REQUIREMENTS, OR THAT THE OPERATION OF THE SOFTWARE WILL

BE UNINTERRUPTED OR ERROR-FREE, OR THAT DEFECTS IN THE SOFTWARE AND THE FONTS WILL BE CORRECTED. FURTHERMORE, METROWERKS DOES NOT WARRANT OR MAKE ANY REPRESENTATIONS REGARDING THE USE OR THE RESULTS OF THE USE OF THE SOFTWARE AND FONTS OR RELATED DOCUMENTATION IN TERMS OF THEIR CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE. NO ORAL OR WRITTEN INFORMATION OR ADVICE GIVEN BY METROWERKS OR A METROWERKS AUTHORIZED REPRESENTATIVE SHALL CREATE A WARRANTY OR IN ANY WAY INCREASE THE SCOPE OF THIS WARRANTY. SHOULD THE SOFTWARE PROVE DEFECTIVE, YOU (AND NOT METROWERKS OR A METROWERKS AUTHORIZED REPRESENTATIVE) ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU.

8. **Limitation of Liability.** UNDER NO CIRCUMSTANCES INCLUDING NEGLIGENCE, SHALL METROWERKS BE LIABLE FOR ANY INCIDENTAL, SPECIAL OR CONSEQUENTIAL DAMAGES THAT RESULT FROM THE USE OR INABILITY TO USE THE SOFTWARE OR RELATED DOCUMENTATION, EVEN IF METROWERKS OR A METROWERKS AUTHORIZED REPRESENTATIVE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. SOME JURISDICTIONS DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU. In no event shall Metrowerks' total liability to you for all damages, losses, and causes of action (whether in contract, tort (including negligence) or otherwise) exceed the portion of the amount paid by you which is fairly attributable to the Software and fonts.

9. **Controlling Law and Severability.** This License shall be governed by and construed in accordance with the laws of the United States and the State of California, as applied to agreements entered into and to be performed entirely within California between California residents. If for any reason a court of competent jurisdiction finds any provision of this License, or portion thereof, to be unenforceable, that provision of the License shall be enforced to the maximum extent permissible so as to effect the intent of the parties, and the remainder of this License shall continue in full force and effect.

10. **Complete Agreement.** This License constitutes the entire agreement between the parties with respect to the use of the Software, the related documentation and fonts, and supersedes all prior or contemporaneous understandings or agreements, written or oral, regarding such subject matter. No amendment to or modification of this License will be binding unless in writing and signed by a duly authorized representative of Metrowerks.

"METROWERKS AND METROWERKS' LICENSOR(S), AND THEIR DIRECTORS, OFFICERS, EMPLOYEES OR AGENTS (COLLECTIVELY METROWERKS) MAKE NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, REGARDING THE SOFTWARE. METROWERKS DOES NOT WARRANT, GUARANTEE OR MAKE ANY REPRESENTATIONS REGARDING THE USE OR THE RESULTS OF THE USE OF THE SOFTWARE IN TERMS OF ITS CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS OR OTHERWISE. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE SOFTWARE IS ASSUMED BY YOU. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED BY SOME JURISDICTIONS. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU."

"IN NO EVENT WILL METROWERKS AND METROWERKS' LICENSOR(S), AND THEIR DIRECTORS, OFFICERS, EMPLOYEES OR AGENTS (COLLECTIVELY METROWERKS) BE LIABLE TO YOU FOR ANY CONSEQUENTIAL, INCIDENTAL OR INDIRECT DAMAGES (INCLUDING DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, AND THE LIKE) ARISING OUT OF THE USE OR INABILITY TO USE THE SOFTWARE EVEN IF METROWERKS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. BECAUSE SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES, THE ABOVE LIMITATIONS MAY NOT APPLY TO YOU. Metrowerks' liability to you for actual damages from any cause whatsoever, and regardless of the form of the action (whether in contract, tort (including negligence), product liability or otherwise), will be limited to exceed the cost of the replacement of the media on which the software is distributed."

This book is dedicated to a select and august group—the hundred or so engineers who, over time and space, have worked to bring CodeWarrior to vibrant life. Most especially, this book is dedicated to the small team that brought CodeWarrior to Windows: Cam Vien, Eric Cloninger, Bob Kushlis, Kurt Ostfeld, Nalinin Prakash, Andrew Southwick, Joel Sumner, and HongGang Zhang. You are, each and all, Code Warriors!

What's on the CD-ROM

This CD-ROM contains two groups of items: CodeWarrior Lite for Windows and example code for this book. You'll find complete instructions on how to install CodeWarrior Lite in Chapter 2. CodeWarrior Lite includes:

- CodeWarrior for Windows—a demo version with compilers and linkers for Windows, Java, and Mac OS
- MWDebug—the CodeWarrior debugger
- Constructor—an interface builder for Mac OS applications
- Win32 x86 Support—files required for Windows programming
- Java Support—files required for Java programming
- Mac OS Support—files required for Mac OS programming
- Metrowerks Standard Library—standard C and C++ libraries
- Examples—megabytes of code for Windows and Java programming
- Docs—some of CodeWarrior's official documentation

To use CodeWarrior (either full or Lite) you need a 486 or better processor (Pentium-class recommended), 16 megabytes of RAM (24 recommended), and Microsoft Windows 95 or Windows NT 4.0 as an operating system.

The example code included on this CD-ROM contains everything you need to successfully complete the Hands-On exercises throughout the book, including CodeWarrior project files, source files, and header files. The examples use either C or Java as a language, as appropriate.

If you own the full version of CodeWarrior for Windows, then you should use it rather than the Lite version. The Lite version does *not* let you create a new project or file, add or remove files from an existing project, or save changes to an existing file. In some exercises you are asked to perform these tasks, which you cannot accomplish using the Lite version.

If you are using CodeWarrior Lite for such an exercise, then you should use the *solution code* rather than the start code. The solution code project has all the necessary files included, and any code changes required in those files. The solution code lets you successfully complete the book without requiring the full version of CodeWarrior.

Free technical support from Metrowerks is provided by acquiring any of Metrowerks' commercial products. You can upgrade from CodeWarrior Lite to a commercial product and thereby benefit from this technical support. See the preceding Metrowerks End-User Software License for more information. Contact Metrowerks at 1-800-377-5416, or via email at **sales@metrowerks.com**.